Help us improve the quality of our code samples by answering this brief survey: http://bit.ly/sample-demo

JavaScript Guide

by 39 contributors: Show all...

Quick Table of Contents

Next »

Help us improve the quality of our code samples by answering this brief survey: http://bit.ly/sample-demo

About this Guide

by 40 contributors: Show all...

JavaScript is a cross-platform, object-based scripting language. This guide explains everything you need to know about using JavaScript.

New features in JavaScript versions

- New in JavaScript 1.2 Redirect 1
- New in JavaScript 1.3 Redirect 1
- New in JavaScript 1.4 Redirect 1
- New in JavaScript 1.5 Redirect 1
- New in JavaScript 1.6 Redirect 1
- New in JavaScript 1.7 Redirect 1
- New in JavaScript 1.8 Redirect 1
- New in JavaScript 1.8.1 Redirect 1
- New in JavaScript 1.8.5 Redirect 1

What you should already know ?

This guide assumes you have the following basic background:

- A general understanding of the Internet and the World Wide Web (WWW).
- Good working knowledge of HyperText Markup Language (HTML).
- Some programming experience. If you are new to programming, try one of the tutorials linked on the page JavaScript

JavaScript versions

Table 1 JavaScript and Navigator versions

JavaScript version	Navigator version

JavaScript 1.0	Navigator 2.0
JavaScript 1.1	Navigator 3.0
JavaScript 1.2	Navigator 4.0-4.05
JavaScript 1.3	Navigator 4.06-4.7x
JavaScript 1.4	
JavaScript 1.5	Navigator 6.0 Mozilla (open source browser)
JavaScript 1.6	Firefox 1.5, other Mozilla 1.8-based products
JavaScript 1.7	Firefox 2, other Mozilla 1.8.1-based products
JavaScript 1.8	Firefox 3, other Gecko 1.9-based products

Where to find JavaScript information

JavaScript documentation includes the following books:

- JavaScript Guide (this guide) provides information about JavaScript language and its objects.
- JavaScript Reference provides reference material for JavaScript language.

If you are new to JavaScript, start with the JavaScript Guide. Once you have a firm grasp of the fundamentals, you can use the JavaScript Reference to get more details on individual objects and statements.

Tips for learning JavaScript

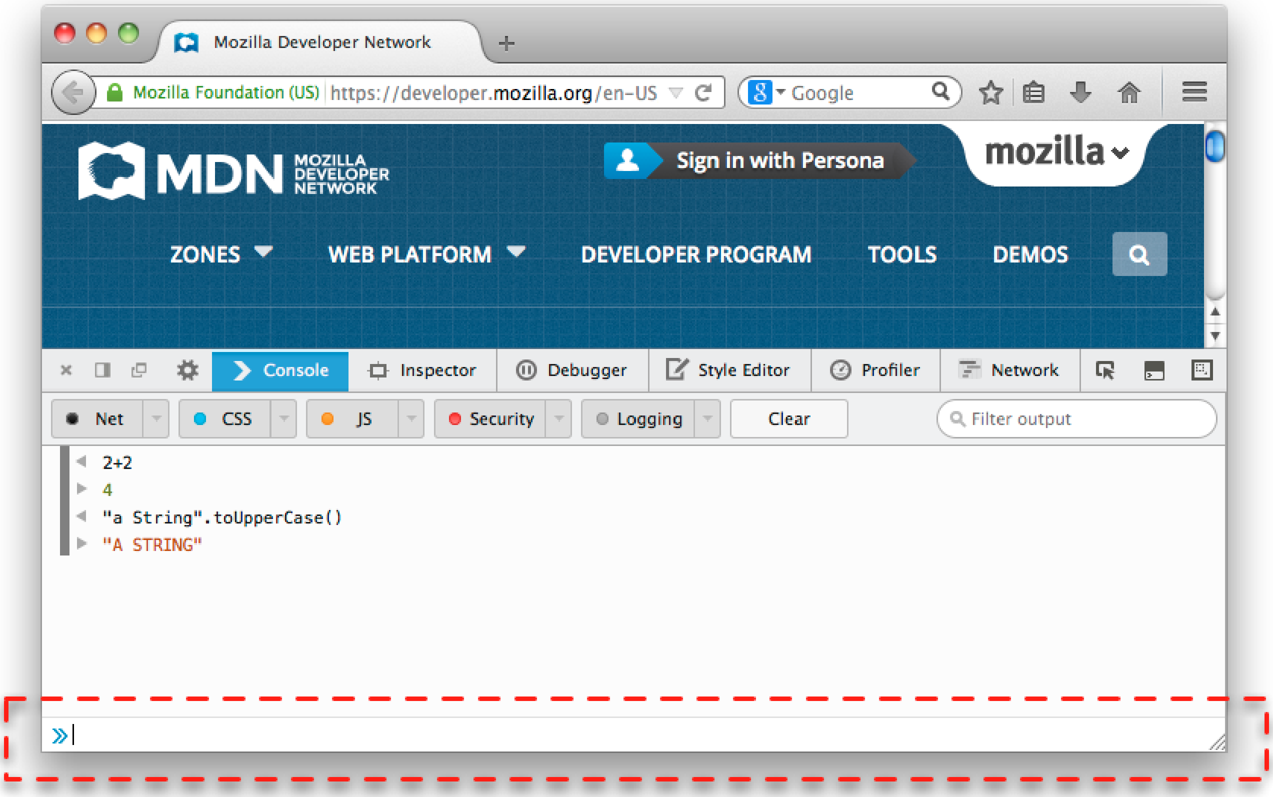
Getting started with JavaScript is easy: all you need is a modern Web browser. This guide includes some JavaScript features which are only currently available in the latest versions of Firefox (and other Gecko powered browsers), so using the most recent version of Firefox is recommended.

There are two tools built into Firefox that are useful for experimenting with JavaScript: the Web Console and Scratchpad.

The Web Console

The Web Console shows you information about the currently loaded Web page, and also includes a command line that you can use to execute JavaScript expressions the current page.

To open the Web Console, select "Web Console" from the "Web Developer" menu, which is under the "Tools" menu in Firefox. It appears at the bottom of the browser window. Along the bottom of the console is a command line that you can use to enter JavaScript, and the output appears in the pane above:



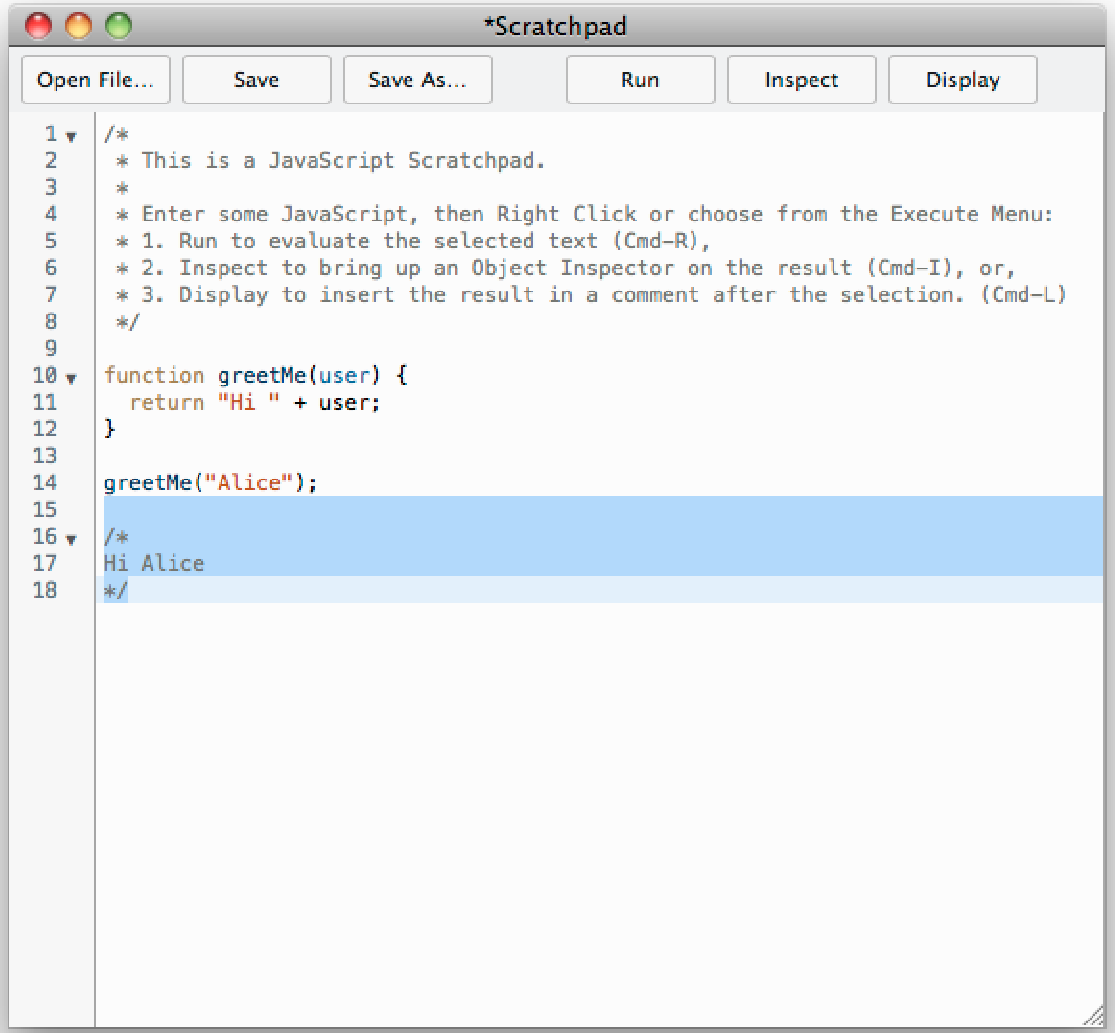
```
2+2
4
"a String".toUpperCase()
"A STRING"
```

Scratchpad

The Web Console is great for executing single lines of JavaScript, but although you can execute multiple lines, it's not very convenient for that, and you can't save your code samples using the Web Console. So for more complex examples Scratchpad is a better tool.

To open Scratchpad, select "Scratchpad" from the "Web Developer" menu, which is under the "Tools" menu in Firefox. It opens in a separate window and is an editor that you can use to write and execute JavaScript in the browser. You can also save scripts to disk and load them from disk.

If you choose "Inspect", the code in your pad is executed in the browser and the result is inserted back into the pad as a comment:



```
/*
 * This is a JavaScript Scratchpad.
 *
 * Enter some JavaScript, then Right Click or choose from the Execute Menu:
 * 1. Run to evaluate the selected text (Cmd-R),
 * 2. Inspect to bring up an Object Inspector on the result (Cmd-I), or,
 * 3. Display to insert the result in a comment after the selection. (Cmd-L)
 */

function greetMe(user) {
  return "Hi " + user;
}

greetMe("Alice");

/*
Hi Alice
*/
```

Document conventions

JavaScript applications run on many operating systems; the information in this book applies to all versions. File and directory paths are given in Windows format (with backslashes separating directory names). For Unix versions, the directory paths are the same, except that you use slashes instead of backslashes to separate directories.

This guide uses uniform resource locators (URLs) of the following form:

`http://`*`server.domain/path/file`*`.html`

In these URLs, *server* represents the name of the server on which you run your application, such as `research1` or `www`; *domain* represents your Internet domain name, such as `netscape.com` or `uiuc.edu`; *path* represents the directory structure on the server; and *file*`.html` represents an individual file name. In general, items in italics in URLs are placeholders and items in normal monospace font are literals. If your server has Secure Sockets Layer (SSL) enabled, you would use `https` instead of `http` in the URL.

This guide uses the following font conventions:

- `The monospace font` is used for sample code and code listings, API and language elements (such as method names and property names), file names, path names, directory names, HTML tags, and any text that must be typed on the screen. (*`Monospace italic font`* is used for placeholders embedded in code.)
- *Italic type* is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
- **Boldface** type is used for glossary terms.

« Previous Next »

Help us improve the quality of our code samples by answering this brief survey: http://bit.ly/sample-demo

JavaScript Overview

This chapter introduces JavaScript and discusses some of its fundamental concepts.

What is JavaScript?

JavaScript is a cross-platform, object-oriented scripting language. JavaScript is a small, lightweight language; it is not useful as a standalone language, but is designed for easy embedding in other products and applications, such as web browsers. Inside a host environment, JavaScript can be connected to the objects of its environment to provide programmatic control over them.

Core JavaScript contains a core set of objects, such as `Array`, `Date`, and `Math`, and a core set of language elements such as operators, control structures, and statements. Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects; for example:

- *Client-side JavaScript* extends the core language by supplying objects to control a browser (Navigator or another web browser) and its Document Object Model (DOM). For example, client-side extensions allow an application to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation.
- *Server-side JavaScript* extends the core language by supplying objects relevant to running JavaScript on a server. For example, server-side extensions allow an application to communicate with a relational database, provide continuity of information from one invocation to another of the application, or perform file manipulations on a server.

Through JavaScript's LiveConnect functionality, you can let Java and JavaScript code communicate with each other. From JavaScript, you can instantiate Java objects and access their public methods and fields. From Java, you can access JavaScript objects, properties, and methods.

Netscape invented JavaScript, and JavaScript was first used in Netscape browsers.

JavaScript and Java

JavaScript and Java are similar in some ways but fundamentally different in some others. The JavaScript language resembles Java but does not have Java's static typing and strong type checking. JavaScript follows most Java expression syntax, naming conventions and basic control-flow constructs which was the reason why it was renamed from LiveScript to JavaScript.

In contrast to Java's compile-time system of classes built by declarations, JavaScript supports a runtime system based on a small number of data types representing numeric, Boolean, and string values. JavaScript has a prototype-based object model instead of the more common class-based object model. The prototype-based model provides dynamic inheritance; that is, what is inherited can vary for individual objects. JavaScript also supports functions without any special declarative requirements. Functions can be properties of objects, executing as loosely typed methods.

JavaScript is a very free-form language compared to Java. You do not have to declare all variables, classes, and methods. You do not have to be concerned with whether methods are public, private, or protected, and you do not have to implement interfaces. Variables, parameters, and function return types are not explicitly typed.

Java is a class-based programming language designed for fast execution and type safety. Type safety means, for instance, that you can't cast a Java integer into an object reference or access private memory by corrupting Java bytecodes. Java's class-based model means that programs consist exclusively of classes and their methods. Java's class inheritance and strong typing generally require tightly coupled object hierarchies. These requirements make Java programming more complex than JavaScript programming.

In contrast, JavaScript descends in spirit from a line of smaller, dynamically typed languages such as HyperTalk and dBASE. These scripting languages offer programming tools to a much wider audience because of their easier syntax, specialized built-in functionality, and minimal requirements for object creation.

Table 1.1 JavaScript compared to Java

JavaScript	Java
Object-oriented. No distinction between types of objects. Inheritance is through the prototype mechanism, and properties and methods can be added to any object dynamically.	Class-based. Objects are divided into classes and instances with all inheritance through the class hierarchy. Classes and instances cannot have properties or methods added dynamically.
Variable data types are not declared (dynamic typing).	Variable data types must be declared (static typing).
Cannot automatically write to hard disk.	Cannot automatically write to hard disk.

For more information on the differences between JavaScript and Java, see the chapter Details of the Object Model.

JavaScript and the ECMAScript Specification

Netscape invented JavaScript, and JavaScript was first used in Netscape browsers. However, Netscape is working with Ecma International — the European association for standardizing information and communication systems (ECMA was formerly an acronym for the European Computer Manufacturers Association) to deliver a standardized, international programming language based on core JavaScript. This standardized version of JavaScript, called ECMAScript, behaves the same way in all applications that support the standard. Companies can use the open standard language to develop their implementation of JavaScript. The ECMAScript standard is documented in the ECMA-262 specification.

The ECMA-262 standard is also approved by the ISO (International Organization for Standardization) as ISO-16262. You can find a PDF version of ECMA-262 (outdated version) at the Mozilla website. You can also find the specification on the Ecma International website. The ECMAScript specification does not describe the Document Object Model (DOM), which is standardized by the World Wide Web Consortium (W3C). The DOM defines the way in which HTML document objects are exposed to your script.

Relationship between JavaScript Versions and ECMAScript Editions

Netscape worked closely with Ecma International to produce the ECMAScript Specification (ECMA-262). The following table describes the relationship between JavaScript versions and ECMAScript editions.

Table 1.2 JavaScript versions and ECMAScript editions

JavaScript version	Relationship to ECMAScript edition
JavaScript 1.1	ECMA-262, Edition 1 is based on JavaScript 1.1.
JavaScript 1.2	ECMA-262 was not complete when JavaScript 1.2 was released. JavaScript 1.2 is not fully compatible with ECMA-262, Edition 1, for the following reasons: • Netscape developed additional features in JavaScript 1.2 that were not considered for ECMA-262. • ECMA-262 adds two new features: internationalization using Unicode, and uniform behavior across all platforms. Several features of JavaScript 1.2, such as the `Date` object, were platform-dependent and used platform-specific behavior.
JavaScript 1.3	JavaScript 1.3 is fully compatible with ECMA-262, Edition 1. JavaScript 1.3 resolved the inconsistencies that JavaScript 1.2 had with ECMA-262, while keeping all the additional features of JavaScript 1.2 except `==` and `!=`, which were changed to conform with ECMA-262.
JavaScript 1.4	JavaScript 1.4 is fully compatible with ECMA-262, Edition 1. The third version of the ECMAScript specification was not finalized when JavaScript 1.4

	was released.
JavaScript 1.5	JavaScript 1.5 is fully compatible with ECMA-262, Edition 3.

Note: *ECMA-262, Edition 2 consisted of minor editorial changes and bug fixes to the Edition 1 specification. The current release by the TC39 working group of Ecma International is ECMAScript Edition 5.1*

The JavaScript Reference indicates which features of the language are ECMAScript-compliant.

JavaScript will always include features that are not part of the ECMAScript Specification; JavaScript is compatible with ECMAScript, while providing additional features.

JavaScript Documentation versus the ECMAScript Specification

The ECMAScript specification is a set of requirements for implementing ECMAScript; it is useful if you want to determine whether a JavaScript feature is supported in other ECMAScript implementations. If you plan to write JavaScript code that uses only features supported by ECMAScript, then you may need to review the ECMAScript specification.

The ECMAScript document is not intended to help script programmers; use the JavaScript documentation for information on writing scripts.

JavaScript and ECMAScript Terminology

The ECMAScript specification uses terminology and syntax that may be unfamiliar to a JavaScript programmer. Although the description of the language may differ in ECMAScript, the language itself remains the same. JavaScript supports all functionality outlined in the ECMAScript specification.

The JavaScript documentation describes aspects of the language that are appropriate for a JavaScript programmer. For example:

- The Global Object is not discussed in the JavaScript documentation because you do not use it directly. The methods and properties of the Global Object, which you do use, are discussed in the JavaScript documentation but are called top-level functions and properties.
- The no parameter (zero-argument) constructor with the `Number` and `String` objects is not discussed in the JavaScript documentation, because what is generated is of little use. A `Number` constructor without an argument returns +0, and a `String` constructor without an argument returns "" (an empty string).

« Previous Next »

Help us improve the quality of our code samples by answering this brief survey: http://bit.ly/sample-demo

Values, variables, and literals

This chapter discusses values that JavaScript recognizes and describes the fundamental building blocks of JavaScript expressions: variables, constants, and literals.

Values

JavaScript recognizes the five following types of primitive values:

Type	Examples of typed values / Notes
Numbers	42, 3.14159
Logical (Boolean)	true / false
Strings	"Howdy"
`null`	a special keyword denoting a null value; `null` is also a primitive value. Because JavaScript is case-sensitive, `null` is not the same as `Null`, `NULL`, or any other variant
`undefined`	a top-level property whose value is undefined; `undefined` is also a primitive value.

Although these data types are a relatively small amount, they enable you to perform useful functions with your applications.

Objects and functions are the other fundamental elements in the language. You can think of objects as named containers for values, and functions as procedures that your application can perform.

Data type conversion

JavaScript is a dynamically typed language. That means you don't have to specify the data type of a variable when you declare it, and data types are converted automatically as needed during script execution. So, for example, you could define a variable as follows:

```
var answer = 42;
```

And later, you could assign the same variable a string value, for example:

```
answer = "Thanks for all the fish...";
```

Because JavaScript is dynamically typed, this assignment does not cause an error message.

In expressions involving numeric and string values with the + operator, JavaScript converts numeric values to strings. For example, consider the following statements:

```
x = "The answer is " + 42 // "The answer is 42"y = 42 + " is the answer"
 // "42 is the answer"
```

In statements involving other operators, JavaScript does not convert numeric values to strings. For example:

```
"37" - 7 // 30"37" + 7 // "377"

```

Converting strings to numbers

In the case that a value representing a number is in memory as a string, there are methods for conversion.

`parseInt()` and `parseFloat()`

See: `parseInt()` and `parseFloat()` pages.

`parseInt` will only return whole numbers, so its use is diminished for decimals. Additionally, a best practice for `parseInt` is to always include the radix parameter. The radix parameter is used to specify which numerical system is to be used.

Unary plus operator

An alternative method of retrieving a number from a string is with the + (unary plus) operator.

```
"1.1" + "1.1" = "1.11.1"
(+"1.1") + (+"1.1") = 2.2
```

```
    // Note: the parentheses are added for clarity, not required.
```

Variables

You use variables as symbolic names for values in your application. The names of variables, called *identifiers*, conform to certain rules.

A JavaScript identifier must start with a letter, underscore (_), or dollar sign ($); subsequent characters can also be digits (0-9). Because JavaScript is case sensitive, letters include the characters "A" through "Z" (uppercase) and the characters "a" through "z" (lowercase).

Starting with JavaScript 1.5, you can use ISO 8859-1 or Unicode letters such as å and ü in identifiers. You can also use the \uXXXX Unicode escape sequences as characters in identifiers.

Some examples of legal names are `Number_hits`, `temp99`, and `_name`.

Declaring variables

You can declare a variable in two ways:

- With the keyword var. For example, `var x = 42`. This syntax can be used to declare both local and global variables.
- By simply assigning it a value. For example, `x = 42`. This always declares a global variable and cannot be changed at the local level. It generates a strict JavaScript warning. You shouldn't use this variant.

Evaluating variables

A variable declared using the `var` statement with no initial value specified has the value `undefined`.

An attempt to access an undeclared variable will result in a `ReferenceError` exception being thrown:

```
var a;
console.log("The value of a is " + a); // logs "The value of a is undefined"consol
 // throws ReferenceError exception
```

You can use `undefined` to determine whether a variable has a value. In the following code, the variable `input` is not assigned a value, and the `if` statement evaluates to `true`.

```
var input;
if(input === undefined){
  doThis();
} else {
  doThat();
}
```

The `undefined` value behaves as `false` when used in a boolean context. For example, the following code executes the function `myFunction` because the `myArray` element is not defined:

```
var myArray = new Array();
if (!myArray[0]) myFunction();
```

The `undefined` value converts to `NaN` when used in numeric context.

```
var a;
a + 2 = NaN
```

When you evaluate a null variable, the null value behaves as 0 in numeric contexts and as false in boolean contexts. For example:

```
var n = null;
console.log(n * 32); // Will log 0 to the console
```

Variable scope

When you declare a variable outside of any function, it is called a *global* variable, because it is available to any other code in the current document. When you declare a variable within a function, it is called a *local* variable, because it is available only within that function.

JavaScript does not have block statement scope; rather, a variable declared within a block is local to the *function (or global scope)* that the block resides within. For example the following code will log `5`, because the scope of `x` is the function (or global context) within which `x` is declared, not the block, which in this case is an `if` statement.

```
if (true) {
  var x = 5;
}
```

```
console.log(x);
```

Another unusual thing about variables in JavaScript is that you can refer to a variable declared later, without getting an exception. This concept is known as hoisting; variables in JavaScript are in a sense "hoisted" or lifted to the top of the function or statement. However, variables that aren't initialized yet will return a value of `undefined`.

```
/**
 * Example 1
 */
console.log(x === undefined); // logs "true"var x = 3;

/**
 * Example 2
 */         // will return a value of undefinedvar myvar = "my value";

(function() {
  console.log(myvar); // undefined  var myvar = "local value";
})();

```

The above examples will be interpreted the same as:

```
/**
 * Example 1
 */
var x;
console.log(x === undefined); // logs "true"x = 3;

/**
 * Example 2
 */
var myvar = "my value";

(function() {
  var myvar;
  console.log(myvar); // undefined  myvar = "local value";
})();

```

Because of hoisting, all `var` statements in a function should be placed as near to the top of the function as possible. This best practice increases the clarity of the code.

Global variables

Global variables are in fact properties of the *global object*. In web pages the global object is `window`, so you can set and access global variables using the `window.`*`variable`* syntax.

Consequently, you can access global variables declared in one window or frame from another window or frame by specifying the window or frame name. For example, if a variable called `phoneNumber` is declared in a `FRAMESET` document, you can refer to this variable from a child frame as `parent.phoneNumber`.

Constants

You can create a read-only, named constant with the `const` keyword. The syntax of a constant identifier is the same as for a variable identifier: it must start with a letter, underscore or dollar sign and can contain alphabetic, numeric, or underscore characters.

```
const prefix = '212';
```

A constant cannot change value through assignment or be re-declared while the script is running.

The scope rules for constants are the same as those for variables, except that the `const` keyword is always required, even for global constants. If the keyword is omitted, the identifier is assumed to represent a variable.

You cannot declare a constant with the same name as a function or variable in the same scope. For example:

```
// THIS WILL CAUSE AN ERRORfunction f() {};
const f = 5;

// THIS WILL CAUSE AN ERROR ALSOfunction f() {
  const g = 5;
  var g;

  //statements}

```

Literals

You use literals to represent values in JavaScript. These are fixed values, not variables, that you *literally* provide in your script. This section describes the following types of literals:

Array literals

An array literal is a list of zero or more expressions, each of which represents an array element, enclosed in square brackets ([]). When you create an array using an array literal, it is initialized with the specified values as its elements, and its length is set to the number of arguments specified.

The following example creates the `coffees` array with three elements and a length of three:

```
var coffees = ["French Roast", "Colombian", "Kona"];
```

Note An array literal is a type of object initializer. See Using Object Initializers.

If an array is created using a literal in a top-level script, JavaScript interprets the array each time it evaluates the expression containing the array literal. In addition, a literal used in a function is created each time the function is called.

Array literals are also `Array` objects. See Array Object for details on `Array` objects.

Extra commas in array literals

You do not have to specify all elements in an array literal. If you put two commas in a row, the array is created with `undefined` for the unspecified elements. The following example creates the `fish` array:

```
var fish = ["Lion", , "Angel"];
```

This array has two elements with values and one empty element (`fish[0]` is "Lion", `fish[1]` is `undefined`, and `fish[2]` is "Angel").

If you include a trailing comma at the end of the list of elements, the comma is ignored. In the following example, the length of the array is three. There is no `myList[3]`. All other commas in the list indicate a new element. (**Note:** trailing commas can create errors in older browser versions and it is a best practice to remove them.)

```
var myList = ['home', , 'school', ];
```

In the following example, the length of the array is four, and `myList[0]` and `myList[2]` are missing.

```
var myList = [ , 'home', , 'school'];
```

In the following example, the length of the array is four, and `myList[1]` and `myList[3]` are missing. Only the last comma is ignored.

```
var myList = ['home', , 'school', , ];
```

Understanding the behavior of extra commas is important to understanding JavaScript as a language, however when writing your own code: explicitly declaring the missing elements as `undefined` will increase your code's clarity and maintainability.

Boolean literals

The Boolean type has two literal values: `true` and `false`.

Do not confuse the primitive Boolean values `true` and `false` with the true and false values of the Boolean object. The Boolean object is a wrapper around the primitive Boolean data type. See Boolean Object for more information.

Integers

Integers can be expressed in decimal (base 10), hexadecimal (base 16), and octal (base 8).

- Decimal integer literal consists of a sequence of digits without a leading 0 (zero).
- Leading 0 (zero) on an integer literal indicates it is in octal. Octal integers can include only the digits 0-7.
- Leading 0x (or 0X) indicates hexadecimal. Hexadecimal integers can include digits (0-9) and the letters a-f and A-F.

Octal integer literals are deprecated and have been removed from the ECMA-262, Edition 3 standard (in *strict mode*). JavaScript 1.5 still supports them for backward compatibility.

Some examples of integer literals are:

```
0, 117 and -345 (decimal, base 10)
015, 0001 and -077 (octal, base 8)
0x1123, 0x00111 and -0xF1A7 (hexadecimal, "hex" or base 16)
```

Floating-point literals

A floating-point literal can have the following parts:

- A decimal integer which can be signed (preceded by "+" or "-"),
- A decimal point ("."),
- A fraction (another decimal number),
- An exponent.

The exponent part is an "e" or "E" followed by an integer, which can be signed (preceded by "+" or "-"). A floating-point literal must have at least one digit and either a decimal point or "e" (or "E").

Some examples of floating-point literals are 3.1415, -3.1E12, .1e12, and 2E-12.

More succinctly, the syntax is:

```
[(+|-)][digits][.digits][(E|e)[(+|-)]digits]
```

For example:

```
3.14
2345.789
.3333333333333333333
-.283185307179586
```

Object literals

An object literal is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({}). You should not use an object literal at the beginning of a statement. This will lead to an error or not behave as you expect, because the { will be interpreted as the beginning of a block.

The following is an example of an object literal. The first element of the `car` object defines a property, `myCar`, and assigns to it a new string, "`Saturn`"; the second element, the `getCar` property, is immediately assigned the result of invoking the function `(CarTypes("Honda"));` the third element, the `special` property, uses an existing variable (`Sales`).

```
var Sales = "Toyota";

function CarTypes(name) {
  if (name == "Honda") {
    return name;
  } else {
    return "Sorry, we don't sell " + name + ".";
  }
}

var car = { myCar: "Saturn", getCar: CarTypes("Honda"), special: Sales };

console.log(car.myCar);   // Saturnconsole.log(car.getCar);  // Hondaconsole.log(
 // Toyota

```

Additionally, you can use a numeric or string literal for the name of a property or nest an object inside another. The following example uses these options.

```
var car = { manyCars: {a: "Saab", "b": "Jeep"}, 7: "Mazda" };

console.log(car.manyCars.b); // Jeepconsole.log(car[7]); // Mazda

```

Object property names can be any string, including the empty string. If the property name would not be a valid JavaScript identifier, it must be enclosed in quotes. Property names that would not be valid identifiers also cannot be accessed as a dot (`.`) property, but can be accessed and set with the array-like notation("`[]`").

```
var unusualPropertyNames = {
  "": "An empty string",
  "!": "Bang!"
}
console.log(unusualPropertyNames."");   // SyntaxError: Unexpected stringconsole.log(unusual
 // "Bang!"

```

Please note:

```
var foo = {a: "alpha", 2: "two"};
console.log(foo.a);    // alphaconsole.log(foo[2]);   // two//console.log(foo.2);
 // two

```

String literals

A string literal is zero or more characters enclosed in double (") or single (') quotation marks. A string must be delimited by quotation marks of the same type; that is, either both single quotation marks or both double quotation marks. The following are examples of string literals:

- `"foo"`
- `'bar'`
- `"1234"`
- `"one line \n another line"`
- `"John's cat"`

You can call any of the methods of the String object on a string literal value—JavaScript automatically converts the string literal to a temporary String object, calls the method, then discards the temporary String object. You can also use the `String.length` property with a string literal:

```
console.log("John's cat".length)
 // Will print the number of symbols in the string including whitespace. In this ca
```

You should use string literals unless you specifically need to use a String object. See String Object for details on `String` objects.

Using special characters in strings

In addition to ordinary characters, you can also include special characters in strings, as shown in the following example.

```
"one line \n another line"
```

The following table lists the special characters that you can use in JavaScript strings.

Table 2.1 JavaScript special characters

Character	Meaning
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Tab
`\v`	Vertical tab
`\'`	Apostrophe or single quote
`\"`	Double quote
`\\`	Backslash character
`\`*`XXX`*	The character with the Latin-1 encoding specified by up to three octal digits *XXX* between 0 and 377. For example, \251 is the octal sequence for the copyright symbol.
`\x`*`XX`*	The character with the Latin-1 encoding specified by the two hexadecimal digits *XX* between 00 and FF. For example, \xA9 is the hexadecimal sequence for the copyright symbol.
`\u`*`XXXX`*	The Unicode character specified by the four hexadecimal digits *XXXX*. For example, \u00A9 is the Unicode sequence for the copyright symbol. See Unicode escape sequences.

Escaping characters

For characters not listed in Table 2.1, a preceding backslash is ignored, but this usage is deprecated and should be avoided.

You can insert a quotation mark inside a string by preceding it with a backslash. This is known as *escaping* the quotation mark. For example:

```
var quote = "He read \"The Cremation of Sam McGee\" by R.W. Service.";
console.log(quote);
```

The result of this would be:

```
He read "The Cremation of Sam McGee" by R.W. Service.
```

To include a literal backslash inside a string, you must escape the backslash character. For example, to assign the file path `c:\temp` to a string, use the following:

```
var home = "c:\\temp";
```

You can also escape line breaks by preceding them with backslash. The backslash and line break are both removed from the value of the string.

```
var str = "this string \
is broken \
across multiple\
lines."
console.log(str);   // this string is broken across multiplelines.
```

Although JavaScript does not have "heredoc" syntax, you can get close by adding a linebreak escape and an escaped linebreak at the end of each line:

```
var poem =
"Roses are red,\n\
Violets are blue.\n\
I'm schizophrenic,\n\
And so am I."
```

Unicode

Unicode is a universal character-coding standard for the interchange and display of principal written languages. It covers the languages of the Americas, Europe, Middle East, Africa, India, Asia, and Pacifica, as well as historic scripts and technical symbols. Unicode allows for the exchange, processing, and display of multilingual texts, as well as the use of common technical and mathematical symbols. It hopes to resolve internationalization problems of multilingual computing, such as different national character standards. Not all modern or archaic scripts, however, are currently supported.

The Unicode character set can be used for all known encoding. Unicode is modeled after the ASCII (American Standard Code for Information Interchange) character set. It uses a numerical value and name for each character. The character encoding specifies the identity of the character and its numeric value (code position), as well as the representation of this value in bits. The 16-bit numeric value (code value) is defined by a hexadecimal number and a prefix U, for example, U+0041 represents A. The unique name for this value is LATIN CAPITAL LETTER A.

Unicode is not supported in versions of JavaScript prior to 1.3.

Unicode compatibility with ASCII and ISO

Unicode is fully compatible with the International Standard ISO/IEC 10646-1; 1993, which is a subset of ISO 10646.

Several encoding standards (including UTF-8, UTF-16 and ISO UCS-2) are used to physically represent Unicode as actual bits.

The UTF-8 encoding of Unicode is compatible with ASCII characters and is supported by many programs. The first 128 Unicode characters correspond to the ASCII characters and have the same byte value. The Unicode characters U+0020 through U+007E are equivalent to the ASCII characters 0x20 through 0x7E. Unlike ASCII, which supports the Latin alphabet and uses a 7-bit character set, UTF-8 uses between one and four octets for each character ("octet" meaning a byte, or 8 bits.) This allows for several million characters. An alternative encoding standard, UTF-16, uses two octets to represent Unicode characters. An escape sequence allows UTF-16 to represent the whole Unicode range by using four octets. The ISO UCS-2 (Universal Character Set) uses two octets.

JavaScript and Navigator support for UTF-8/Unicode means you can use non-Latin, international, and localized characters, plus special technical symbols in JavaScript programs. Unicode provides a standard way to encode multilingual text. Since the UTF-8 encoding of Unicode is compatible with ASCII, programs can use ASCII characters. You can use non-ASCII Unicode characters in the comments, string literals, identifiers, and regular expressions of JavaScript.

Unicode escape sequences

You can use the Unicode escape sequence in string literals, regular expressions, and identifiers. The escape sequence consists of six ASCII characters: \u and a four-digit hexadecimal number. For example, \u00A9 represents the copyright symbol. Every Unicode escape sequence in JavaScript is interpreted as one character.

The following code returns the copyright symbol and the string "Netscape Communications".

```
var x = "\u00A9 Netscape Communications";
```

The following table lists frequently used special characters and their Unicode value.

Table 2.2 Unicode values for special characters

Category	Unicode value	Name	Format name
White space values	\u0009	Tab	<TAB>
	\u000B	Vertical Tab	<VT>
	\u000C	Form Feed	<FF>
	\u0020	Space	<SP>
Line terminator values	\u000A	Line Feed	<LF>
	\u000D	Carriage Return	<CR>
Additional Unicode escape sequence values	\u0008	Backspace	<BS>
	\u0009	Horizontal Tab	<HT>
	\u0022	Double Quote	"
	\u0027	Single Quote	'
	\u005C	Backslash	\

The JavaScript use of the Unicode escape sequence is different from Java. In JavaScript, the escape sequence is never interpreted as a special character first. For example, a line terminator escape sequence inside a string does not terminate the string before it is interpreted by the function. JavaScript ignores any escape sequence if it is used in comments. In Java, if an escape sequence is used in a single comment line, it is interpreted as an Unicode character. For a string literal, the Java compiler interprets the escape sequences first. For example, if a line terminator escape character (e.g., \u000A) is used in Java, it terminates the string literal. In Java, this leads to an error, because line terminators are not allowed in string literals. You must use \n for a line feed in a string literal. In JavaScript, the escape sequence works the same way as \n.

Unicode characters in JavaScript files

Earlier versions of Gecko assumed the Latin-1 character encoding for JavaScript files loaded from XUL. Starting with Gecko 1.8, the character encoding is inferred from the XUL file's encoding. Please see International characters in XUL JavaScript for more information.

Displaying characters with Unicode

You can use Unicode to display the characters in different languages or technical symbols. For characters to be displayed properly, a client such as Mozilla Firefox or Netscape needs to support Unicode. Moreover, an appropriate Unicode font must be available to the client, and the client platform must support Unicode. Often, Unicode fonts do not display all the Unicode characters. Some platforms, such as Windows 95, provide partial support for Unicode.

To receive non-ASCII character input, the client needs to send the input as Unicode. Using a standard enhanced keyboard, the client cannot easily input the additional characters supported by Unicode. Sometimes, the only way to input Unicode characters is by using Unicode escape sequences.

For more information on Unicode, see the Unicode Home Page and The Unicode Standard, Version 2.0, published by Addison-Wesley, 1996.

Resources

- Text Escaping and Unescaping in JavaScript – an utility to convert characters in JavaScript unicode values

« Previous Next »

Help us improve the quality of our code samples by answering this brief survey: http://bit.ly/sample-demo

Expressions and operators

by 26 contributors: Show all...

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

Expressions

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

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

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

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

JavaScript has the following expression categories:

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

Operators

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

- Assignment operators
- Comparison operators

- Arithmetic operators
- Bitwise operators
- Logical operators
- String operators
- Special operators

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

```
operand1 operator operand2
```

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

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

```
operator operand
```

or

```
operand operator
```

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

Assignment operators

An assignment operator assigns a value to its left operand based on the value of its right operand. The 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 shorthand for the operations listed in the following table:

Table 3.1 Assignment operators

Shorthand operator	Meaning
`x += y`	`x = x + y`

`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`
`x <<= y`	`x = x << y`
`x >>= y`	`x = x >> y`
`x >>>= y`	`x = x >>> y`
`x &= y`	`x = x & y`
`x ^= y`	`x = x ^ y`
`x |= y`	`x = x | y`

Comparison operators

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

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

Table 3.2 Comparison operators

Operator	Description	Examples returning true

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

		`var1 != 4`

Not equal (`!=`)	Returns true if the operands are not equal.	`var2 != "3"`
Strict equal (`===`)	Returns true if the operands are equal and of the same type. See also `Object.is` and sameness in JS.	`3 === var1`
Strict not equal (`!==`)	Returns true if the operands are not equal and/or not of the same type.	`var1 !== "3"` `3 !== '3'`
Greater than (`>`)	Returns true if the left operand is greater than the right operand.	`var2 > var1` `"12" > 2`
Greater than or equal (`>=`)	Returns true if the left operand is greater than or equal to the right operand.	`var2 >= var1` `var1 >= 3`
Less than (`<`)	Returns true if the left operand is less than the right operand.	`var1 < var2` `"2" < "12"`
Less than or equal (`<=`)	Returns true if the left operand is less than or equal to the right operand.	`var1 <= var2` `var2 <= 5`

Arithmetic operators

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

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

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

Table 3.3 Arithmetic operators

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

Bitwise operators

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

The following table summarizes JavaScript's bitwise operators.

Table 3.4 Bitwise operators

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

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

Bitwise logical operators

Conceptually, the bitwise logical operators work as follows:

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

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

Table 3.5 Bitwise operator examples

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

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

Bitwise shift operators

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

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

The shift operators are listed in the following table.

Table 3.6 Bitwise shift operators

Operator	Description	Example
`<<` (Left shift)	This operator shifts the first operand the specified number of bits to the left. Excess bits shifted off to the left are discarded. Zero bits are shifted in from the right.	`9<<2` yields 36, because 1001 shifted 2 bits to the left becomes 100100, which is 36.
`>>` (Sign-propagating right shift)	This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Copies of the leftmost bit are shifted in from the left.	`9>>2` yields 2, because 1001 shifted 2 bits to the right becomes 10, which is 2. Likewise, `-9>>2` yields -3, because the sign is preserved.
`>>>` (Zero-fill right shift)	This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left.	`19>>>2` yields 4, because 10011 shifted 2 bits to the right becomes 100, which is 4. For non-negative numbers, zero-fill right shift and sign-propagating right shift yield the same result.

Logical operators

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

Table 3.6 Logical operators

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

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

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

```
var a1 =  true && true;     // t && t returns truevar a2 =  true && false;    //
 // t && f returns false
```

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

```
var o1 =  true || true;     // t || t returns truevar o2 = false || true;     //
 // t || f returns Cat
```

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

```
var n1 = !true;  // !t returns falsevar n2 = !false; // !f returns truevar n3 = !
 // !t returns false
```

Short-circuit evaluation

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

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

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

String operators

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

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

Special operators

JavaScript provides the following special operators:

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

Conditional operator

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

```
condition ? val1 : val2
```

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

For example,

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

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

Comma operator

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

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

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

`delete`

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

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

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

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

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

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

```
x = 42;
var y = 43;
myobj = new Number();
myobj.h = 4;    // create property hdelete x;       // returns true (can delete i
 // returns true (can delete if declared implicitly)

```

Deleting array elements

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

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

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

```

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

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

```

`in`

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

```
propNameOrNumber in objectName
```

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

The following examples show some uses of the `in` operator.

```
// Arraysvar trees = new Array("redwood", "bay", "cedar", "oak", "maple");

0 in trees;        // returns true3 in trees;        // returns true6 in trees;

"length" in myString;  // returns true// Custom objectsvar mycar = {make: "Honda'

```

```
"make" in mycar;  // returns true"model" in mycar; // returns true
```

instanceof

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

```
objectName instanceof objectType
```

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

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

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

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

new

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

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

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

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

this

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

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

```
this.propertyName
```

Example 1.
Suppose a function called `validate` validates an object's `value` property, given the object and the high and low values:

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

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

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

Example 2.
When combined with the `form` property, `this` can refer to the current object's parent form. In the following example, the form `myForm` contains a `Text` object and a button. When the user clicks the button, the value of the `Text` object is set to the form's name. The button's `onClick` event handler uses `this.form` to refer to the parent form, `myForm`.

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

`typeof`

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

```
typeof operand
```

```
typeof (operand)
```

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

Suppose you define the following variables:

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

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

```
typeof myFun;     // returns "function"typeof shape;     // returns "string"typeo
 // returns "undefined"

```

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

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

```

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

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

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

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

```

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

```
typeof blur;        // returns "function"typeof eval;        // returns "functior
 // returns "function"

```

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

```
typeof Date;     // returns "function"typeof Function; // returns "function"typec
 // returns "function"

```

`void`

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

```
void (expression)
```

```
void expression
```

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

You can use the `void` operator to specify an expression as a hypertext link. The expression is evaluated but is not loaded in place of the current document.

The following code creates a hypertext link that does nothing when the user clicks it. When the user clicks the link, `void(0)` evaluates to undefined, which has no effect in JavaScript.

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

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

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

Operator precedence

The *precedence* of operators determines the order they are applied when evaluating an expression. You can override operator precedence by using parentheses.

The following table describes the precedence of operators, from highest to lowest.

*In accordance with relevant discussion, this table was reversed to list operators in **decreasing** order of priority.*

Table 3.7 Operator precedence

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

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

A more detailed version of this table, complete with links to additional details about each operator, may be found in JavaScript Reference.

« Previous Next »

Help us improve the quality of our code samples by answering this brief survey: http://bit.ly/sample-demo

Regular Expressions

by 40 contributors: Show all...

Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects. These patterns are used with the `exec` and `test` methods of `RegExp`, and with the `match`, `replace`, `search`, and `split` methods of `String`. This chapter describes JavaScript regular expressions.

Creating a Regular Expression

You construct a regular expression in one of two ways:

- Using a regular expression literal, as follows:

  ```
  var re = /ab+c/;
  ```

  Regular expression literals provide compilation of the regular expression when the script is loaded. When the regular expression will remain constant, use this for better performance.

- Calling the constructor function of the `RegExp` object, as follows:

  ```
  var re = new RegExp("ab+c");
  ```

  Using the constructor function 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.

Writing a Regular Expression Pattern

A regular expression pattern is composed of simple characters, such as `/abc/`, or a combination of simple and special characters, such as `/ab*c/` or `/Chapter (\d+)\.\d*/`. The last example includes parentheses which are used as a memory device. The match made with this part of the pattern is remembered for later use, as described in Using Parenthesized Substring Matches.

Using Simple Patterns

Simple patterns are constructed of characters for which you want to find a direct match. For example, the pattern `/abc/` matches character combinations in strings only when exactly the characters 'abc' occur together and in that order. Such a match would succeed in the strings "Hi, do you know your abc's?" and "The latest airplane designs evolved from slabcraft." In both cases the match is with the substring 'abc'. There is no match in the string 'Grab crab' because while it contains the substring 'ab c', it does not contain the exact substring 'abc'.

Using Special Characters

When the search for a match requires something more than a direct match, such as finding one or more b's, or finding white space, the pattern includes special characters. For example, the pattern `/ab*c/` matches any character combination in which a single 'a' is followed by zero or more 'b's (`*` means 0 or more occurrences of the preceding item) and then immediately followed by 'c'. In the string "cbbabbbbcdebc," the pattern matches the substring 'abbbbc'.

The following table provides a complete list and description of the special characters that can be used in regular expressions.

Table 4.1 Special characters in regular expressions.

Character	Meaning
`\`	Matches according to the following rules: A backslash that precedes a non-special character indicates that the next character is special and is not to be interpreted literally. For example, a '`b`' without a preceding '\' generally matches lowercase 'b's wherever they occur. But a '`\b`' by itself doesn't match any character; it forms the special word boundary character. A backslash that precedes a special character indicates that the next character is not special and should be interpreted literally. For example, the pattern `/a*/` relies on the special character '`*`' to match 0 or more a's. By contrast, the pattern `/a\*/` removes the specialness of the '`*`' to enable matches with strings like 'a*'. Do not forget to escape \ itself while using the RegExp("pattern") notation because \ is also an escape character in strings.
`^`	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 'A' in "An E".

	The '^' has a different meaning when it appears as the first character in a character set pattern. See complemented character sets for details and an example.
`$`	Matches end of input. If the multiline flag is set to true, also matches immediately before a line break character. For example, `/t$/` does not match the 't' in "eater", but does match it in "eat".
`*`	Matches the preceding character 0 or more times. Equivalent to {0,}. For example, `/bo*/` matches 'boooo' in "A ghost booooed" and 'b' in "A bird warbled", but nothing in "A goat grunted".
`+`	Matches the preceding character 1 or more times. Equivalent to {1,}. For example, `/a+/` matches the 'a' in "candy" and all the a's in "caaaaaaandy".
`?`	Matches the preceding character 0 or 1 time. Equivalent to `{0,1}`. For example, `/e?le?/` matches the 'el' in "angel" and the 'le' in "angle" and also the 'l' in "oslo". If used immediately after any of the quantifiers *, +, ?, or {}, makes the quantifier non-greedy (matching the fewest possible characters), as opposed to the default, which is greedy (matching as many characters as possible). For example, applying `/\d+/` to "123abc" matches "123". But applying `/\d+?/` to that same string matches only the "1". Also used in lookahead assertions, as described in the `x(?=y)` and `x(?!y)` entries of this table.
`.`	(The decimal point) matches any single character except the newline character. For example, `/.n/` matches 'an' and 'on' in "nay, an apple is on the tree", but not 'nay'.
`(x)`	Matches 'x' and remembers the match, as the following example shows. The parentheses are called *capturing parentheses*. The '`(foo)`' and '`(bar)`' in the pattern `/(foo) (bar) \1 \2/` match and remember the first two words in the string "foo bar foo bar". The `\1` and `\2` in the pattern match

	the string's last two words. Note that `\1, \2, \n` are used in the matching part of the regex. In the replacement part of a regex the syntax `$1, $2, $n` must be used, e.g.: `'bar foo'.replace( /(...) (...)/, '$2 $1' ).`
`(?:x)`	Matches 'x' but does not remember the match. The parentheses are called *non-capturing parentheses*, and let you define subexpressions for regular expression operators to work with. Consider the sample expression `/(?:foo){1,2}/`. Without the non-capturing parentheses, the `{1,2}` characters would apply only to the last 'o' in 'foo'. With the non-capturing parentheses, the `{1,2}` applies to the entire word 'foo'.
`x(?=y)`	Matches 'x' only if 'x' is followed by 'y'. This is called a lookahead. 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'. This is called a negated lookahead. For example, `/\d+(?!\.)/` matches a number only if it is not followed by a decimal point. The regular expression `/\d+(?!\.)/.exec("3.141")` matches '141' but not '3.141'.
`x|y`	Matches either 'x' or 'y'. For example, `/green|red/` matches 'green' in "green apple" and 'red' in "red apple."
`{n}`	Matches exactly n occurrences of the preceding character. N must be a positive integer. For example, `/a{2}/` doesn't match the 'a' in "candy," but it does match all of the a's in "caandy," and the first two a's in "caaandy."
`{n,m}`	Where `n` and `m` are positive integers. Matches at least `n` and at most `m` occurrences of the preceding character. When `m` is zero, it can be omitted. For example, `/a{1,3}/` matches nothing in "cndy", the 'a' in "candy," the first 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.

`[xyz]`	Character set. This pattern type matches any one of the characters in the brackets, including escape sequences. Special characters like the dot(`.`) and asterisk (`*`) are not special inside a character set, so they don't need to be escaped. You can specify a range of characters by using a hyphen, as the following examples illustrate. The pattern `[a-d]`, which performs the same match as `[abcd]`, matches the 'b' in "brisket" and the 'c' in "city". The patterns `/[a-z.]+/` and `/[\w.]+/` match the entire string "test.i.ng".
`[^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. Everything that works in the normal character set also works here. For example, `[^abc]` is the same as `[^a-c]`. They initially match 'r' in "brisket" and 'h' in "chop."
`[\b]`	Matches a backspace (U+0008). You need to use square brackets if you want to match a literal backspace character. (Not to be confused with `\b`.)
`\b`	Matches a word boundary. A word boundary matches the position where a word character is not followed or preceeded by another word-character. Note that a matched word boundary is not included in the match. In other words, the length of a matched word boundary is zero. (Not to be confused with `[\b]`.) Examples: `/\bm/` matches the 'm' in "moon" ; `/oo\b/` does not match the 'oo' in "moon", because 'oo' is followed by 'n' which is a word character; `/oon\b/` matches the 'oon' in "moon", because 'oon' is the end of the string, thus not followed by a word character; `/\w\b\w/` will never match anything, because a word character can never be followed by both a non-word and a word character.
`\B`	Matches a non-word boundary. This matches a position where the previous and next character are of the same type: Either both must be words, or both must be non-words. The beginning and end of a string are considered non-words. For example, `/\B../` matches 'oo' in "noonday" (, and `/y\B./` matches 'ye' in "possibly yesterday."

`\cX`	Where *X* is a character ranging from A to Z. Matches a control character in a string. For example, `/\cM/` matches control-M (U+000D) in a string.
`\d`	Matches a digit character. Equivalent to `[0-9]`. For example, `/\d/` or `/[0-9]/` matches '2' in "B2 is the suite number."
`\D`	Matches any non-digit character. Equivalent to `[^0-9]`. For example, `/\D/` or `/[^0-9]/` matches 'B' in "B2 is the suite number."
`\f`	Matches a form feed (U+000C).
`\n`	Matches a line feed (U+000A).
`\r`	Matches a carriage return (U+000D).
`\s`	Matches a single white space character, including space, tab, form feed, line feed. Equivalent to `[ \f\n\r\t\v\u00a0\u1680\u180e\u2000\u2001\u2002\u2003\u2004\u2005\u2006\u2007\u2008\u2009\u200a\u2028\u2029\u202f\u205f\u3000]`. For example, `/\s\w*/` matches ' bar' in "foo bar."
`\S`	Matches a single character other than white space. Equivalent to `[^ \f\n\r\t\v\u00a0\u1680\u180e\u2000\u2001\u2002\u2003\u2004\u2005\u2006\u2007\u2008\u2009\u200a\u2028\u2029\u202f\u205f\u3000]`. For example, `/\S\w*/` matches 'foo' in "foo bar."
`\t`	Matches a tab (U+0009).
`\v`	Matches a vertical tab (U+000B).
`\w`	Matches any alphanumeric character 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 non-word character. Equivalent to `[^A-Za-z0-9_]`. For example, `/\W/` or `/[^A-Za-z0-9_]/` matches '%' in "50%."
`\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."
`\0`	Matches a NULL (U+0000) character. Do not follow this with another digit, because `\0<digits>` is an octal escape sequence.
`\xhh`	Matches the character with the code hh (two hexadecimal digits)
`\uhhhh`	Matches the character with the code hhhh (four hexadecimal digits).

Escaping user input to be treated as a literal string within a regular expression can be accomplished by simple replacement:

```
function escapeRegExp(string){
  return string.replace(/([.*+?^=!:${}()|\[\]\/\\])/g, "\\$1");
}
```

Using Parentheses

Parentheses around any part of the regular expression pattern cause that part of the matched substring to be remembered. Once remembered, the substring can be recalled for other use, as described in Using Parenthesized Substring Matches.

For example, the pattern `/Chapter (\d+)\.\d*/` illustrates additional escaped and special characters and indicates that part of the pattern should be remembered. It matches precisely the characters 'Chapter ' followed by one or more numeric characters (`\d` means any numeric character and `+` means 1 or more times), followed by a decimal point (which in itself is a special character; preceding the decimal point with \ means the pattern must look for the literal character '.'), followed by any numeric character 0 or more times (`\d` means numeric character, `*` means 0 or more times). In addition, parentheses are used to remember the first matched numeric characters.

This pattern is found in "Open Chapter 4.3, paragraph 6" and '4' is remembered. The pattern is not found in "Chapter 3 and 4", because that string does not have a period after the '3'.

To match a substring without causing the matched part to be remembered, within the parentheses preface the pattern with `?:`. For example, `(?:\d+)` matches one or more numeric characters but does not remember the matched characters.

Working with Regular Expressions

Regular expressions are used with the `RegExp` methods `test` and `exec` and with the `String` methods `match`, `replace`, `search`, and `split`. These methods are explained in detail in the JavaScript Reference.

Table 4.2 Methods that use regular expressions

Method	Description
`exec`	A `RegExp` method that executes a search for a match in a string. It returns an array of information.
`test`	A `RegExp` method that tests for a match in a string. It returns true or false.
`match`	A `String` method that executes a search for a match in a string. It returns an array of information or null on a mismatch.
`search`	A `String` method that tests for a match in a string. It returns the index of the match, or -1 if the search fails.
`replace`	A `String` method that executes a search for a match in a string, and replaces the matched substring with a replacement substring.
`split`	A `String` method that uses a regular expression or a fixed string to break a string into an array of substrings.

When you want to know whether a pattern is found in a string, use the `test` or `search` method; for more information (but slower execution) use the `exec` or `match` methods. If you use `exec` or `match` and if the match succeeds, these methods return an array and update properties of the associated regular expression object and also of the predefined regular expression object, `RegExp`. If the match fails, the `exec` method returns `null` (which coerces to `false`).

In the following example, the script uses the `exec` method to find a match in a string.

```
var myRe = /d(b+)d/g;
var myArray = myRe.exec("cdbbdbsbz");
```

If you do not need to access the properties of the regular expression, an alternative way of creating `myArray` is with this script:

```
var myArray = /d(b+)d/g.exec("cdbbdbsbz");
```

If you want to construct the regular expression from a string, yet another alternative is this script:

```
var myRe = new RegExp("d(b+)d", "g");
var myArray = myRe.exec("cdbbdbsbz");
```

With these scripts, the match succeeds and returns the array and updates the properties shown in the following table.

Table 4.3 Results of regular expression execution.

Object	Property or index	Description	In this example
`myArray`		The matched string and all remembered substrings.	`["dbbd", "bb"]`
	`index`	The 0-based index of the match in the input string.	`1`
	`input`	The original string.	`"cdbbdbsbz"`
	`[0]`	The last matched characters.	`"dbbd"`
`myRe`	`lastIndex`	The index at which to start the next match. (This property is set only if the regular expression uses the g option, described in Advanced Searching With Flags.)	`5`
	`source`	The text of the pattern. Updated at the time that the regular expression is created, not executed.	`"d(b+)d"`

As shown in the second form of this example, you can use a regular expression created with an object initializer without assigning it to a variable. If you do, however, every occurrence is a new regular expression. For this reason, if you use this form without assigning it to a variable, you cannot subsequently access the properties of that regular expression. For example, assume you have this script:

```
var myRe = /d(b+)d/g;
var myArray = myRe.exec("cdbbdbsbz");
console.log("The value of lastIndex is " + myRe.lastIndex);
```

This script displays:

```
The value of lastIndex is 5
```

However, if you have this script:

```
var myArray = /d(b+)d/g.exec("cdbbdbsbz");
console.log("The value of lastIndex is " + /d(b+)d/g.lastIndex);
```

It displays:

```
The value of lastIndex is 0
```

The occurrences of `/d(b+)d/g` in the two statements are different regular expression objects and hence have different values for their `lastIndex` property. If you need to access the properties of a regular expression created with an object initializer, you should first assign it to a variable.

Using Parenthesized Substring Matches

Including parentheses in a regular expression pattern causes the corresponding submatch to be remembered. For example, `/a(b)c/` matches the characters 'abc' and remembers 'b'. To recall these parenthesized substring matches, use the `Array` elements `[1]`, ..., `[n]`.

The number of possible parenthesized substrings is unlimited. The returned array holds all that were found. The following examples illustrate how to use parenthesized substring matches.

Example 1

The following script uses the `replace()` method to switch the words in the string. For the replacement text, the script uses the `$1` and `$2` in the replacement to denote the first and second parenthesized substring matches.

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

This prints "Smith, John".

Advanced Searching With Flags

Regular expressions have four optional flags that allow for global and case insensitive searching. These flags can be used separately or together in any order, and are included as part of the regular expression.

Table 4.4 Regular expression flags.

Flag	Description
`g`	Global search.
i	Case-insensitive search.
m	Multi-line search.
y	Perform a "sticky" search that matches starting at the current position in the target string.

Firefox 3 note

Support for the `y` flag was added in Firefox 3. The `y` flag fails if the match doesn't succeed at the current position in the target string.

To include a flag with the regular expression, use this syntax:

```
var re = /pattern/flags;
```

or

```
var re = new RegExp("pattern", "flags");
```

Note that the flags are an integral part of a regular expression. They cannot be added or removed later.

For example, `re = /\w+\s/g` creates a regular expression that looks for one or more characters followed by a space, and it looks for this combination throughout the string.

```
var re = /\w+\s/g;
var str = "fee fi fo fum";
var myArray = str.match(re);
console.log(myArray);
```

This displays ["fee ", "fi ", "fo "]. In this example, you could replace the line:

```
var re = /\w+\s/g;
```

with:

```
var re = new RegExp("\\w+\\s", "g");
```

and get the same result.

The `m` flag is used to specify that a multiline input string should be treated as multiple lines. If the `m` flag is used, `^` and `$` match at the start or end of any line within the input string instead of the start or end of the entire string.

Examples

The following examples show some uses of regular expressions.

Changing the Order in an Input String

The following example illustrates the formation of regular expressions and the use of `string.split()` and `string.replace()`. It cleans a roughly formatted input string containing names (first name first) separated by blanks, tabs and exactly one semicolon. Finally, it reverses the name order (last name first) and sorts the list.

```
// The name string contains multiple spaces and tabs,// and may have multiple spa

var output = ["---------- Original String\n", names + "\n"];

// Prepare two regular expression patterns and array storage.// Split the string

// Break the string into pieces separated by the pattern above and// store the pi

// new pattern: one or more characters then spaces then characters.// Use parenth

// New array for holding names being processed.var bySurnameList = [];

// Display the name array and populate the new array// with comma-separated names
output.push("---------- After Split by Regular Expression");

var i, len;
for (i = 0, len = nameList.length; i < len; i++){
  output.push(nameList[i]);
  bySurnameList[i] = nameList[i].replace(pattern, "$2, $1");
}

// Display the new array.output.push("---------- Names Reversed");
for (i = 0, len = bySurnameList.length; i < len; i++){
  output.push(bySurnameList[i]);
}

// Sort by last name, then display the sorted array.bySurnameList.sort();
output.push("---------- Sorted");
for (i = 0, len = bySurnameList.length; i < len; i++){
  output.push(bySurnameList[i]);
}

output.push("---------- End");

console.log(output.join("\n"));

```

Using Special Characters to Verify Input

In the following example, the user is expected to enter a phone number. When the user presses the "Check" button, the script checks the validity of the number. If the number is valid (matches the character sequence specified by the regular expression), the script shows a message thanking the user and confirming the number. If the number is invalid, the script informs the user that the phone number is not valid.

Within non-capturing parentheses `(?:` , the regular expression looks for three numeric characters `\d{3}` OR `|` a left parenthesis `\(` followed by three digits `\d{3}`, followed by a close parenthesis `\)`, (end non-capturing parenthesis `)`), followed by one dash, forward slash, or decimal point and when found, remember the character `([-\/\.])`, followed by three digits `\d{3}`, followed by the remembered match of a dash, forward slash, or decimal point `\1`, followed by four digits `\d{4}`.

The `Change` event activated when the user presses Enter sets the value of `RegExp.input`.

```
<!DOCTYPE html>
<html>
```

```
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  <meta http-equiv="Content-Script-Type" content="text/javascript">
  <script type="text/javascript">
     var re = /(?:\d{3}|\(\d{3}\))([-\/\.])\d{3}\1\d{4}/;
     function testInfo(phoneInput){
      var OK = re.exec(phoneInput.value);
      if (!OK)
       window.alert(RegExp.input + " isn't a phone number with area code!")
      else
       window.alert("Thanks, your phone number is " + OK[0]);
     }
    </script>
 </head>
 <body>
  <p>Enter your phone number (with area code) and then click "Check".
    <br>The expected format is like ###-###-####.</p>
  <form action="#">
   <input id="phone"><button onclick="testInfo(document.getElementById('phone')
  </form>
 </body>
</html>
```

« Previous Next »

Help us improve the quality of our code samples by answering this brief survey: http://bit.ly/sample-demo

Statements (Control Flow)

by 19 contributors: Show all...

JavaScript supports a compact set of statements, specifically control flow statements, that you can use to incorporate a great deal of interactivity in Web pages. This chapter provides an overview of these statements.

Any expression is also a statement. See Expressions and Operators for complete information about expressions.

Use the semicolon (`;`) character to separate statements in JavaScript code.

See the JavaScript Reference for details about the statements in this chapter.

Block Statement

A block statement is used to group statements. The block is delimited by a pair of curly brackets:

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

Example

Block statements are commonly used with control flow statements (e.g. `if`, `for`, `while`).

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

Here, `{ x++; }` is the block statement.

Important: JavaScript does **not** have block scope. Variables introduced within 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;
}
alert(x); // outputs 2
```

This outputs 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.

Conditional Statements

A conditional statement is a set of commands that executes if a specified condition is true. JavaScript supports two conditional statements: `if...else` and `switch`.

if...else Statement

Use the `if` statement to execute a statement if a logical condition is true. Use the optional `else` clause to execute a statement if the condition is false. An `if` statement looks as follows:

```
if (condition)
  statement_1
[else
  statement_2]
```

`condition` can be any expression that evaluates to true or false. See Boolean for an explanation of what evaluates to `true` and `false`. If `condition` evaluates to true, `statement_1` is executed; otherwise, `statement_2` is executed. `statement_1` and `statement_2` can be any statement, including further nested `if` statements.

You may also compound the statements using `else if` to have multiple conditions tested in sequence, as follows:

```
if (condition)
  statement_1
[else if (condition_2)
  statement_2]
...
[else if (condition_n_1)
  statement_n_1]
[else
  statement_n]
```

To execute multiple statements, group them within a block statement (`{ ... }`) . In general, it's good practice to always use block statements, especially when nesting `if` statements:

```
if (condition) {
    statement_1_runs_if_condition_is_true
    statement_2_runs_if_condition_is_true
} else {
    statement_3_runs_if_condition_is_false
    statement_4_runs_if_condition_is_false
}
```

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 */
}
```

The following values will evaluate to false:

- `false`
- `undefined`

- `null`
- `0`
- `NaN`
- the empty string (`""`)

All other values, including all objects evaluate to true when passed to a conditional statement.

Do not confuse the primitive boolean values `true` and `false` with the true and false values of the Boolean object. For example:

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

Example

In the following example, the function `checkData` returns true if the number of characters in a `Text` object is three; otherwise, it displays an alert and returns false.

```
function checkData() {
  if (document.form1.threeChar.value.length == 3) {
    return true;
  } else {
    alert("Enter exactly three characters. " +
      document.form1.threeChar.value + " is not valid.");
    return false;
  }
}
```

switch Statement

A `switch` statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, the program executes the associated statement. A `switch` statement looks as follows:

```
switch (expression) {
   case label_1:
      statements_1
      [break;]
   case label_2:
      statements_2
      [break;]
   ...
   default:
      statements_def
      [break;]
}
```

The program first looks for a `case` clause with a label matching the value of expression and then transfers control to that clause, executing the associated statements. If no matching label 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` clause 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.

Example

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

```
switch (fruittype) {
   case "Oranges":
      document.write("Oranges are $0.59 a pound.<br>");
      break;
   case "Apples":
      document.write("Apples are $0.32 a pound.<br>");
      break;
   case "Bananas":
      document.write("Bananas are $0.48 a pound.<br>");
      break;
   case "Cherries":
      document.write("Cherries are $3.00 a pound.<br>");
      break;
   case "Mangoes":
      document.write("Mangoes are $0.56 a pound.<br>");
       break;
   case "Papayas":
      document.write("Mangoes and papayas are $2.79 a pound.<br>");
      break;
   default:
      document.write("Sorry, we are out of " + fruittype + ".<br>");
}
document.write("Is there anything else you'd like?<br>");
```

Loop Statements

A loop is a set of commands that executes repeatedly until a specified condition is met. JavaScript supports the `for`, `do while`, and `while` loop statements, as well as label (label is not itself a looping statement, but is frequently used with these statements). In addition, you can use the `break` and `continue` statements within loop statements.

Another statement, `for...in`, executes statements repeatedly but is used for object manipulation. See Object Manipulation Statements.

The loop statements are:

- for Statement
- do...while Statement
- while Statement
- label Statement
- break Statement
- continue Statement

for Statement

A `for` loop repeats until a specified condition evaluates to false. The JavaScript for loop is similar to the Java and C `for` loop. A `for` statement looks as follows:

```
for ([initialExpression]; [condition]; [incrementExpression])
   statement
```

When a `for` loop executes, the following occurs:

1. The initializing expression `initialExpression`, if any, is executed. This expression usually initializes one or more loop counters, but the syntax allows an expression of any degree of complexity. This expression can also declare variables.

2. The `condition` expression is evaluated. If the value of `condition` is true, the loop statements execute. If the value of `condition` is false, the `for` loop terminates. If the `condition` expression is omitted entirely, the condition is assumed to be true.
3. The `statement` executes. To execute multiple statements, use a block statement (`{ ... }`) to group those statements.
4. The update expression `incrementExpression`, if there is one, executes, and control returns to step 2.

Example

The following function contains a `for` statement that counts the number of selected options in a scrolling list (a `Select` object that allows multiple selections). The `for` statement declares the variable `i` and initializes it to zero. It checks that `i` is less than the number of options in the `Select` object, performs the succeeding `if` statement, and increments `i` by one after each pass through the loop.

```
<script>

function howMany(selectObject) {
   var numberSelected = 0;
   for (var i = 0; i < selectObject.options.length; i++) {
      if (selectObject.options[i].selected)
         numberSelected++;
   }
   return numberSelected;
}

</script>

<form name="selectForm">
   <p>
      <strong>Choose some music types, then click the button below:</strong>
      <br/>
      <select name="musicTypes" multiple="multiple">
         <option selected="selected">R&B</option>
         <option>Jazz</option>
         <option>Blues</option>
         <option>New Age</option>
         <option>Classical</option>
         <option>Opera</option>
      </select>
   </p>
   <p>
      <input type="button" value="How many are selected?"
            onclick="alert ('Number of options selected: ' + howMany(document.se
   </p>
```

```
</form>
```

do...while Statement

The `do...while` statement repeats until a specified condition evaluates to false. A `do...while` statement looks as follows:

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

`statement` executes once before the condition is checked. To execute multiple statements, use a block statement (`{ ... }`) to group those statements. If `condition` is true, the statement executes again. At the end of every execution, the condition is checked. When the condition is false, execution stops and control passes to the statement following `do...while`.

Example

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

```
do {
   i += 1;
   document.write(i);
} while (i < 5);
```

while Statement

A `while` statement executes its statements as long as a specified condition evaluates to true. A `while` statement looks as follows:

```
while (condition)
   statement
```

If the condition becomes false, `statement` within the loop stops executing and control passes to the statement following the loop.

The condition test occurs before `statement` in the loop are executed. If the condition returns true, `statement` is executed and the condition is tested again. If the condition returns false, execution stops and control is passed to the statement following `while`.

To execute multiple statements, use a block statement ({ ... }) to group those statements.

Example 1
The following `while` loop iterates as long as `n` is less than three:

```
n = 0;
x = 0;
while (n < 3) {
   n++;
   x += n;
}
```

With each iteration, the loop increments `n` and adds that value 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.

Example 2
Avoid infinite loops. Make sure the condition in a loop eventually becomes false; otherwise, the loop will never terminate. The statements in the following `while` loop execute forever because the condition never becomes false:

```
while (true) {
   alert("Hello, world");
}
```

label Statement

A label provides a statement with an identifier that lets you refer to it elsewhere in your program. For example, you can use a label to identify a loop, and then use the `break` or `continue` statements to indicate whether a program should interrupt the loop or continue its execution.

The syntax of the label statement looks like the following:

```
label :
   statement
```

The value of *`label`* may be any JavaScript identifier that is not a reserved word. The *`statement`* that you identify with a label may be any statement.

Example
In this example, the label `markLoop` identifies a `while` loop.

```
markLoop:
while (theMark == true) {
   doSomething();
}
```

break Statement

Use the `break` statement to terminate a loop, `switch`, or in conjunction with a label statement.

- When you use `break` without a label, it terminates the innermost enclosing `while`, `do-while`, `for`, or `switch` immediately and transfers control to the following statement.
- When you use `break` with a label, it terminates the specified labeled statement.

The syntax of the `break` statement looks like this:

1. `break;`
2. `break` *`label`*`;`

The first form of the syntax terminates the innermost enclosing loop or `switch`; the second form of the syntax terminates the specified enclosing label statement.

Example 1:
The following example iterates through the elements in an array until it finds the index of an element whose value is `theValue`:

```
for (i = 0; i < a.length; i++) {
    if (a[i] == theValue)
      break;
}
```

Example 2: Breaking to a Label

```
var x = 0;
var z = 0
labelCancelLoops: while (true) {
    console.log("Outer loops: " + x);
    x += 1;
    z = 1;
    while (true) {
        console.log("Inner loops: " + z);
        z += 1;
        if (z === 10 && x === 10) {
            break labelCancelLoops;
        } else if (z === 10) {
            break;
        }
    }
}
```

continue Statement

The `continue` statement can be used to restart a `while`, `do-while`, `for`, or `label` statement.

- When you use `continue` without a label, it terminates the current iteration of the innermost enclosing `while`, `do-while` or `for` statement and continues execution of the loop with the next iteration. In contrast to the `break` statement, `continue` does not terminate the execution of the loop entirely. In a `while` loop, it jumps back to the condition. In a `for` loop, it jumps to the `increment-expression`.
- When you use `continue` with a label, it applies to the looping statement identified with that label.

The syntax of the `continue` statement looks like the following:

1. `continue;`
2. `continue` *`label;`*

Example 1

The following example shows a `while` loop with a `continue` statement that executes when the value of `i` is three. Thus, `n` takes on the values one, three, seven, and twelve.

```
i = 0;
n = 0;
while (i < 5) {
   i++;
   if (i == 3)
      continue;
   n += i;
}
```

Example 2

A statement labeled `checkiandj` contains a statement labeled `checkj`. If `continue` is encountered, the program terminates the current iteration of `checkj` and begins the next iteration. Each time `continue` is encountered, `checkj` reiterates until its condition returns `false`. When `false` is returned, the remainder of the `checkiandj` statement is completed, and `checkiandj` reiterates until its condition returns `false`. When `false` is returned, the program continues at the statement following `checkiandj`.

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

```
checkiandj :
   while (i < 4) {
      document.write(i + "<br/>");
      i += 1;
      checkj :
         while (j > 4) {
            document.write(j + "<br/>");
            j -= 1;
            if ((j % 2) == 0)
               continue checkj;
            document.write(j + " is odd.<br/>");
         }
      document.write("i = " + i + "<br/>");
      document.write("j = " + j + "<br/>");
   }
```

Object Manipulation Statements

JavaScript uses the `for...in`, `for each...in`, and `with` statements to manipulate objects.

for...in Statement

The `for...in` statement iterates a specified variable over all the properties of an object. For each distinct property, JavaScript executes the specified statements. A `for...in` statement looks as follows:

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

Example

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

```
function dump_props(obj, obj_name) {
   var result = "";
   for (var i in obj) {
      result += obj_name + "." + i + " = " + obj[i] + "<br>";
   }
   result += "<hr>";
   return result;
}
```

For an object `car` with properties `make` and `model`, `result` would be:

```
car.make = Ford
car.model = Mustang
```

Arrays

Although it may be tempting to use this as a way to iterate over Array elements, because the **for...in** statement iterates over user-defined properties in addition to the array elements, if you modify the Array object, such as adding custom properties or methods, the **for...in** statement will return the name of your user-defined properties in addition to the numeric indexes. Thus it is better to use a traditional for loop with a numeric index when iterating over arrays.

for each...in Statement

`for each...in` is a loop statement introduced in JavaScript 1.6. It is similar to `for...in`, but iterates over the values of object's properties, not their names.

```
var sum = 0;
var obj = {prop1: 5, prop2: 13, prop3: 8};
for each (var item in obj) {
  sum += item;
}
print(sum); // prints "26", which is 5+13+8
```

Comments

Comments are author notations that explain what a script does. Comments are ignored by the interpreter. JavaScript supports Java and C++-style comments:

- Comments on a single line are preceded by a double-slash (//).
- Comments that span multiple lines are preceded by /* and followed by */:

Example

The following example shows two comments:

```
// This is a single-line comment.
/* This is a multiple-line comment. It can be of any length, and
you can put whatever you want here. */

```

Exception Handling Statements

You can throw exceptions using the `throw` statement and handle them using the `try...catch` statements.

You can also use the `try...catch` statement to handle Java exceptions (though there is a bug 391642 with this). See Handling Java Exceptions in JavaScript and JavaScript to Java Communication for information.

- throw Statement
- try...catch Statement

Exception Types

Just about any object can be thrown in JavaScript. Nevertheless, not all thrown objects are created equal. While it is fairly common to throw numbers or strings as errors it is frequently more effective to use one of the exception types specifically created for this purpose:

- ECMAScript exceptions
- DOMException
- nsIXPCException (XPConnect)

throw Statement

Use the `throw` statement to throw an exception. When you throw an exception, you specify the expression containing the value to be thrown:

```
throw expression;
```

You may throw any expression, not just expressions of a specific type. The following code throws several exceptions of varying types:

```
throw "Error2";   //String typethrow 42;         //Number typethrow true;       /
;

```

***Note:** 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 `myUserException` of type `UserException` and uses it in a throw statement.*

```
// Create an object type UserExceptionfunction UserException (message){
  this.message=message;
  this.name="UserException";
}

// Make the exception convert to a pretty string when used as a string (e.g. by t
  return this.name + ': "' + this.message + '"';
}

// Create an instance of the object type and throw itthrow new UserException("Val
;

```

try...catch Statement

The `try...catch` statement marks a block of statements to try, and specifies one or more responses should an exception be thrown. If an exception is thrown, the `try...catch` statement catches it.

The `try...catch` statement consists of a `try` block, which contains one or more statements, and zero or more `catch` blocks, containing 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` block. If no exception is thrown in the `try` block, the `catch` block is skipped. The `finally` block executes after the `try` and `catch` blocks execute but before the statements following the `try...catch` statement.

The following example uses a `try...catch` statement. The example calls a function that retrieves a month name from an array based on the value passed to the function. If the value does not correspond to a month number (1-12), an exception is thrown with the value `"InvalidMonthNo"` and the statements in the `catch` block set the `monthName` variable to `unknown`.

```
function getMonthName (mo) {
    mo=mo-1; // Adjust month number for array index (1=Jan, 12=Dec)    var months
          "Aug","Sep","Oct","Nov","Dec");
    if (months[mo] != null) {
       return months[mo]
    } else {
       throw "InvalidMonthNo"          //throw keyword is used here    }
}

try {// statements to try    monthName=getMonthName(myMonth) // function could th
catch (e) {
    monthName="unknown"
    logMyErrors(e) // pass exception object to error handler}

```

The catch Block

You can use a `catch` block to handle all exceptions that may be generated in the `try` block.

```
catch (catchID) {
  statements
}
```

The `catch` block specifies an identifier (`catchID` in the preceding syntax) that holds the value specified by the `throw` statement; you can use this identifier to get information about the exception that was thrown. JavaScript creates this identifier when the `catch` block is entered; the identifier lasts only for the duration of the `catch` block; after the `catch` block finishes executing, the identifier is no longer available.

For example, the following code throws an exception. When the exception occurs, control transfers to the `catch` block.

```
try {
   throw "myException" // generates an exception}

catch (e) {// statements to handle any exceptions   logMyErrors(e) // pass except
}

```

The finally Block

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

You can use the `finally` block 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` block closes the file before the script fails.

```
openMyFile();
try {
    writeMyFile(theData); //This may throw a error}catch(e){
    handleError(e); // If we got a error we handle it}finally {
    closeMyFile(); // always close the resource}

```

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:

```
function f() {
    try {
        alert(0);
        throw "bogus";
    } catch(e) {
        alert(1);
        return true; // this return statement is suspended until finally block ha
```

```
          alert(3);
          return false; // overwrites the previous "return"        alert(4); // not
      // "return false" is executed now
      alert(5); // not reachable}
f(); // alerts 0, 1, 3; returns false

```

Nesting try...catch Statements

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

Utilizing Error objects

Depending on the type of error, you may be able to use the 'name' and 'message' properties to get a more refined message. 'name' provides the general class of Error (e.g., 'DOMException' or 'Error'), while 'message' generally provides a more succinct message than one would get by converting the error object to a string.

If you are throwing your own exceptions, in order to take advantage of these properties (such as if your catch block doesn't discriminate between your own exceptions and system ones), you can use the Error constructor. For example:

```
function doSomethingErrorProne () {
   if (ourCodeMakesAMistake()) {
      throw (new Error('The message'));
   }
   else {
      doSomethingToGetAJavascriptError();
   }
}
....
try {
   doSomethingErrorProne();
}
catch (e) {
   alert(e.name);// alerts 'Error'   alert(e.message); // alerts 'The message' or
}

```

« Previous   Next »

Help us improve the quality of our code samples by answering this brief survey: http://bit.ly/sample-demo

Functions

by 35 contributors: Show all...

This article is in need of a technical review.

This article is in need of an editorial review.

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure —a set of statements that performs a task or calculates a value. To use a function, you must define it somewhere in the scope from which you wish to call it.

Defining functions

A **function definition** (also called a **function declaration**) consists of the `function` keyword, followed by

- The name of the function.
- A list of arguments to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly braces, `{ }`.

For example, the following code defines a simple function named `square`:

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

The function `square` takes one argument, called `number`. The function consists of one statement that says to return the argument of the function (that is, `number`) multiplied by itself. The `return` statement specifies the value returned by the function.

```
return number * number;
```

Primitive parameters (such as a number) are passed to functions **by value**; the value is passed to the function, but if the function changes the value of the parameter, this change is not reflected globally or in the calling function.

If you pass an object (i.e. a non-primitive value, such as `Array` or a user-defined object) as a parameter, and the function changes the object's properties, that change is visible outside the function, as shown in the following example:

```
function myFunc(theObject) {
  theObject.make = "Toyota";
}

var mycar = {make: "Honda", model: "Accord", year: 1998},
    x,
    y;

x = mycar.make;     // x gets the value "Honda"
myFunc(mycar);
y = mycar.make;     // y gets the value "Toyota"
 // (the make property was changed by the function)

```

Note that assigning a new object to the parameter will **not** have any effect outside the function, because this is changing the value of the parameter rather than the value of one of the object's properties:

```
function myFunc(theObject) {
  theObject = {make: "Ford", model: "Focus", year: 2006};
}

var mycar = {make: "Honda", model: "Accord", year: 1998},
    x,
    y;

x = mycar.make;     // x gets the value "Honda"
myFunc(mycar);
y = mycar.make;     // y still gets the value "Honda"

```

While the function declaration above is syntactically a statement, functions can also be created by a **function expression**. Such a function can be **anonymous**; it does not have to have a name. For example, the function `square` could have been defined as:

```
var square = function(number) {return number * number};
var x = square(4) //x gets the value 16
```

However, a name can be provided with a function expression, and can be used inside the function to refer to itself, or in a debugger to identify the function in stack traces:

```
var factorial = function fac(n) {return n<2 ? 1 : n*fac(n-1)};

console.log(factorial(3));
```

Function expressions are convenient when passing a function as an argument to another function. The following example shows a `map` function being defined and then called with an anonymous function as its first parameter:

```
function map(f,a) {
  var result = [], // Create a new Array      i;
  for (i = 0; i != a.length; i++)
    result[i] = f(a[i]);
  return result;
}

```

The following code:

```
map(function(x) {return x * x * x}, [0, 1, 2, 5, 10]);
```

returns [0, 1, 8, 125, 1000].

In JavaScript, a function can be defined based on a condition. For example, the following function definition defines `myFunc` only if `num` equals 0:

```
var myFunc;
if (num == 0){
  myFunc = function(theObject) {
    theObject.make = "Toyota"
  }
}
```

In addition to defining functions as described here, you can also use the `Function` constructor to create functions from a string at runtime, much like `eval()`.

A **method** is a function that is a property of an object. Read more about objects and methods in Working with Objects.

Calling functions

Defining a function does not execute it. Defining the function simply names the function and specifies what to do when the function is called. **Calling** the function actually performs the specified actions with the indicated parameters. For example, if you define the function `square`, you could call it as follows:

```
square(5);
```

The preceding statement calls the function with an argument of 5. The function executes its statements and returns the value 25.

Functions must be in scope when they are called, but the function declaration can be below the call, as in this example:

```
console.log(square(5));
/* ... */
function square(n){return n*n}
```

The scope of a function is the function in which it is declared, or the entire program if it is declared at the top level. Note that this works only when defining the function using the above syntax (i.e. `function funcName(){}`). The code below will not work.

```
console.log(square(5));
square = function (n) {
  return n * n;
}
```

The arguments of a function are not limited to strings and numbers. You can pass whole objects to a function, too. The `show_props` function (defined in Working with Objects) is an example of a function that takes an object as an argument.

A function can be recursive; that is, it can call itself. For example, here is a function that computes factorials recursively:

```
function factorial(n){
  if ((n == 0) || (n == 1))
    return 1;
  else
    return (n * factorial(n - 1));
}
```

You could then compute the factorials of one through five as follows:

```
var a, b, c, d, e;
a = factorial(1); // a gets the value 1b = factorial(2); // b gets the value 2c =
 // e gets the value 120

```

There are other ways to call functions. There are often cases where a function needs to be called dynamically, or the number of arguments to a function vary, or in which the context of the function call needs to be set to a specific object determined at runtime. It turns out that functions are, themselves, objects, and these objects in turn have methods (see the `Function` object). One of these, the `apply()` method, can be used to achieve this goal.

Function scope

Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function. However, a function can access all variables and functions defined inside the scope in which it is defined. In other words, a function defined in the global scope can access all variables defined in the global scope. A function defined inside another function can also access all variables defined in its parent function and any other variable to which the parent function has access.

```
// The following variables are defined in the global scopevar num1 = 20,
    num2 = 3,
    name = "Chamahk";

// This function is defined in the global scopefunction multiply() {
  return num1 * num2;
}


multiply(); // Returns 60// A nested function examplefunction getScore () {
  var num1 = 2,
      num2 = 3;

  function add() {
    return name + " scored " + (num1 + num2);
  }

  return add();
}

getScore(); // Returns "Chamahk scored 5"
```

Closures

Closures are one of the most powerful features of JavaScript. JavaScript allows for the nesting of functions and, in addition, grants the inner function full access to all the variables and functions defined inside the outer function (and all other variables and functions that the outer function has access to). However, the outer function does not have access to the variables and functions defined inside the inner function. This provides a sort of security for the variables of the inner function. Also, since the inner function has access to the scope of the outer function, the variables and functions defined in the outer function will live longer than the outer function itself, if the inner function manages to survive beyond the life of the outer function. A closure is created when the inner function is somehow made available to any scope outside the outer function.

```
var pet = function(name) {          // The outer function defines a variable call
        return name;                // The inner function has access to the "name

      return getName;               // Return the inner function, thereby exposir
    myPet = pet("Vivie");

myPet();                            // Returns "Vivie"
```

It can be much more complex than the code above. An object containing methods for manipulating the inner variables of the outer function can be returned.

```
var createPet = function(name) {
  var sex;

  return {
    setName: function(newName) {
      name = newName;
    },

    getName: function() {
      return name;
    },

    getSex: function() {
      return sex;
    },

    setSex: function(newSex) {
      if(typeof newSex == "string" && (newSex.toLowerCase() == "male" || newSex.t
        sex = newSex;
      }
    }
  }
}

var pet = createPet("Vivie");
pet.getName();                  // Vivie
pet.setName("Oliver");
pet.setSex("male");
pet.getSex();                   // malepet.getName();                  // Oliver

```

In the codes above, the `name` variable of the outer function is accessible to the inner functions, and there is no other way to access the inner variables except through the inner functions. The inner variables of the inner function act as safe stores for the inner functions. They hold "persistent", yet secure, data for the inner functions to work with. The functions do not even have to be assigned to a variable, or have a name.

```
var getCode = (function(){
  var secureCode = "0]Eal(eh&2";    // A code we do not want outsiders to be able
  return function () {
    return secureCode;
  };
})();

getCode();    // Returns the secureCode

```

There are, however, a number of pitfalls to watch out for when using closures. If an enclosed function defines a variable with the same name as the name of a variable in the outer scope, there is no way to refer to the variable in the outer scope again.

```
var createPet = function(name) {  // Outer function defines a variable called "na
    setName: function(name) {    // Enclosed function also defines a variable cal
  }
}

```

The magical `this` variable is very tricky in closures. They have to be used carefully, as what `this` refers to depends completely on where the function was called, rather than where it was defined. An excellent and elaborate article on closures can be found here.

Using the arguments object

The arguments of a function are maintained in an array-like object. Within a function, you can address the arguments passed to it as follows:

```
arguments[i]
```

where `i` is the ordinal number of the argument, starting at zero. So, the first argument passed to a function would be `arguments[0]`. The total number of arguments is indicated by `arguments.length`.

Using the `arguments` object, you can call a function with more arguments than it is formally declared to accept. This is often useful if you don't know in advance how many arguments will be passed to the function. You can use `arguments.length` to determine the number of arguments actually passed to the function, and then access each argument using the `arguments` object.

For example, consider a function that concatenates several strings. The only formal argument for the function is a string that specifies the characters that separate the items to concatenate. The function is defined as follows:

```
function myConcat(separator) {
   var result = "", // initialize list       i;
   // iterate through arguments   for (i = 1; i < arguments.length; i++) {
      result += arguments[i] + separator;
   }
   return result;
}

```

You can pass any number of arguments to this function, and it concatenates each argument into a string "list":

```
// returns "red, orange, blue, "myConcat(", ", "red", "orange", "blue");

// returns "elephant; giraffe; lion; cheetah; "myConcat("; ", "elephant", "giraff

// returns "sage. basil. oregano. pepper. parsley. "myConcat(". ", "sage", "basil
;

```

Please note that the `arguments` variable is "array-like", but not an array. It is array-like in that is has a numbered index and a `length` property. However, it does not possess all of the array-manipulation methods.

See the `Function` object in the JavaScript Reference for more information.

Predefined functions

JavaScript has several top-level predefined functions:

- eval
- isFinite
- isNaN
- parseInt and parseFloat
- Number and String
- encodeURI, decodeURI, encodeURIComponent, and decodeURIComponent (all available with Javascript 1.5 and later).

The following sections introduce these functions. See the JavaScript Reference for detailed information on all of these functions.

eval Function

The `eval` function evaluates a string of JavaScript code without reference to a particular object. The syntax of `eval` is:

```
eval(expr);
```

where `expr` is a string to be evaluated.

If the string represents an expression, `eval` evaluates the expression. If the argument represents one or more JavaScript statements, eval performs the statements. The scope of `eval` code is identical to the scope of the calling code. Do not call `eval` to evaluate an arithmetic expression; JavaScript evaluates arithmetic expressions automatically.

isFinite function

The `isFinite` function evaluates an argument to determine whether it is a finite number. The syntax of `isFinite` is:

```
isFinite(number);
```

where `number` is the number to evaluate.

If the argument is `NaN`, positive infinity or negative infinity, this method returns `false`, otherwise it returns `true`.

The following code checks client input to determine whether it is a finite number.

```
if(isFinite(ClientInput)){
   /* take specific steps */
}
```

isNaN function

The `isNaN` function evaluates an argument to determine if it is "NaN" (not a number). The syntax of `isNaN` is:

```
isNaN(testValue);
```

where `testValue` is the value you want to evaluate.

The `parseFloat` and `parseInt` functions return "NaN" when they evaluate a value that is not a number. `isNaN` returns true if passed "NaN," and false otherwise.

The following code evaluates `floatValue` to determine if it is a number and then calls a procedure accordingly:

```
var floatValue = parseFloat(toFloat);

if (isNaN(floatValue)) {
   notFloat();
} else {
   isFloat();
}
```

parseInt and parseFloat functions

The two "parse" functions, `parseInt` and `parseFloat`, return a numeric value when given a string as an argument.

The syntax of `parseFloat` is:

```
parseFloat(str);
```

where `parseFloat` parses its argument, the string `str`, and attempts to return a floating-point number. If it encounters a character other than a sign (+ or -), a numeral (0-9), a decimal point, or an exponent, then it returns the value up to that point and ignores that character and all succeeding characters. If the first character cannot be converted to a number, it returns "NaN" (not a number).

The syntax of `parseInt` is:

```
parseInt(str [, radix]);
```

`parseInt` parses its first argument, the string `str`, and attempts to return an integer of the specified `radix` (base), indicated by the second, optional argument, `radix`. For example, a radix of ten indicates to convert to a decimal number, eight octal, sixteen hexadecimal, and so on. For radixes above ten, the letters of the alphabet indicate numerals greater than nine. For example, for hexadecimal numbers (base 16), A through F are used.

If `parseInt` encounters a character that is not a numeral in the specified radix, it ignores it and all succeeding characters and returns the integer value parsed up to that point. If the first character cannot be converted to a number in the specified radix, it returns "NaN." The `parseInt` function truncates the string to integer values.

Number and String functions

The `Number` and `String` functions let you convert an object to a number or a string. The syntax of these functions is:

```
var objRef;
objRef = Number(objRef);
objRef = String(objRef);
```

where `objRef` is an object reference. Number uses the valueOf() method of the object; String uses the toString() method of the object.

The following example converts the `Date` object to a readable string.

```
var D = new Date(430054663215),
    x;
x = String(D);
 // x equals "Thu Aug 18 04:37:43 GMT-0700 (Pacific Daylight Time) 1983"
```

The following example converts the `String` object to `Number` object.

```
var str = "12",
    num;
num = Number(str);
```

You can check it. Use DOM method `write()` and JavaScript `typeof` operator.

```
var str = "12",
    num;
document.write(typeof str);
document.write("<br/>");
num = Number(str);
document.write(typeof num);
```

escape and unescape functions(Obsoleted above JavaScript 1.5)

The `escape` and `unescape` functions do not work properly for non-ASCII characters and have been deprecated. In JavaScript 1.5 and later, use `encodeURI`, `decodeURI`, `encodeURIComponent`, and `decodeURIComponent`.

The `escape` and `unescape` functions let you encode and decode strings. The `escape` function returns the hexadecimal encoding of an argument in the ISO Latin character set. The `unescape` function returns the ASCII string for the specified hexadecimal encoding value.

The syntax of these functions is:

```
escape(string);
unescape(string);
```

These functions are used primarily with server-side JavaScript to encode and decode name/value pairs in URLs.

« Previous Next »

Help us improve the quality of our code samples by answering this brief survey: http://bit.ly/sample-demo

Working with objects

by 39 contributors: Show all...

JavaScript is designed on a simple object-based paradigm. An object is a collection of properties, and a property is an association between a name and a value. A property's value can be a function, in which case the property is known as a *method*. In addition to objects that are predefined in the browser, you can define your own objects.

This chapter describes how to use objects, properties, functions, and methods, and how to create your own objects.

Objects overview

Objects in JavaScript, just as many other programming languages, can be compared to objects in real life. The concept of objects in JavaScript can be understood with real life, tangible objects.

In JavaScript, an object is a standalone entity, with properties and type. Compare it with a cup, for example. A cup is an object, with properties. A cup has a color, a design, weight, a material it is made of, etc. The same way, JavaScript objects can have properties, which define their characteristics.

Objects and properties

A JavaScript object has properties associated with it. A property of an object can be explained as a variable that is attached to the object. Object properties are basically the same as ordinary JavaScript variables, except for the attachment to objects. The properties of an object define the characteristics of the object. You access the properties of an object with a simple dot-notation:

```
objectName.propertyName
```

Like all JavaScript variables, both the object name (which could be a normal variable) and property name are case sensitive. You can define a property by assigning it a value. For example, let's create an object named `myCar` and give it properties named `make`, `model`, and `year` as follows:

```
var myCar = new Object();
myCar.make = "Ford";
myCar.model = "Mustang";
myCar.year = 1969;
```

Properties of JavaScript objects can also be accessed or set using a bracket notation. Objects are sometimes called *associative arrays*, since each property is associated with a string value that can be used to access it. So, for example, you could access the properties of the `myCar` object as follows:

```
myCar["make"] = "Ford";
myCar["model"] = "Mustang";
myCar["year"] = 1969;
```

An object property name can be any valid JavaScript string, or anything that can be converted to a string, including the empty string. However, any property name that is not a valid JavaScript identifier (for example, a property name that has a space or a hyphen, or that starts with a number) can only be accessed using the square bracket notation. This notation is also very useful when property names are to be dynamically determined (when the property name is not determined until runtime). Examples are as follows:

```
var myObj = new Object(),
    str = "myString",
    rand = Math.random(),
    obj = new Object();

myObj.type              = "Dot syntax";
myObj["date created"]   = "String with space";
myObj[str]              = "String value";
myObj[rand]             = "Random Number";
myObj[obj]              = "Object";
myObj[""]               = "Even an empty string";

console.log(myObj);
```

You can also access properties by using a string value that is stored in a variable:

```
var propertyName = "make";
myCar[propertyName] = "Ford";

propertyName = "model";
```

```
myCar[propertyName] = "Mustang";
```

You can use the bracket notation with for...in to iterate over all the enumerable properties of an object. To illustrate how this works, the following function displays the properties of the object when you pass the object and the object's name as arguments to the function:

```
function showProps(obj, objName) {
  var result = "";
  for (var i in obj) {
    if (obj.hasOwnProperty(i)) {
        result += objName + "." + i + " = " + obj[i] + "\n";
    }
  }
  return result;
}
```

So, the function call `showProps(myCar, "myCar")` would return the following:

```
myCar.make = Ford
myCar.model = Mustang
myCar.year = 1969
```

Object everything

In JavaScript, almost everything is an object. All primitive types except `null` and `undefined` are treated as objects. They can be assigned properties (assigned properties of some types are not persistent), and they have all characteristics of objects.

Enumerating all properties of an object

Starting with ECMAScript 5, there are three native ways to list/traverse object properties:

- for...in loops
  This method traverses all enumerable properties of an object and its prototype chain
- Object.keys(o)
  This method returns an array with all the own (not in the prototype chain) enumerable properties' names ("keys") of an object `o`.
- Object.getOwnPropertyNames(o)
  This method returns an array containing all own properties' names (enumerable or not) of an

object o.

Before ECMAScript 5, there was no native way to list all properties of an object. However, this can be achieved with the following function:

```
function listAllProperties(o){
    var objectToInspect;
    var result = [];

    for(objectToInspect = o; objectToInspect !== null; objectToInspect = Object.ge
        result = result.concat(Object.getOwnPropertyNames(objectToInspect));
    }

    return result;
}
```

This can be useful to reveal "hidden" properties (properties in the prototype chain which are not accessible through the object, because another property has the same name earlier in the prototype chain). Listing accessible properties only can easily be done by removing duplicates in the array.

Creating new objects

JavaScript has a number of predefined objects. In addition, you can create your own objects. In JavaScript 1.2 and later, you can create an object using an object initializer. Alternatively, you can first create a constructor function and then instantiate an object using that function and the `new` operator.

Using object initializers

In addition to creating objects using a constructor function, you can create objects using an object initializer. Using object initializers is sometimes referred to as creating objects with literal notation. "Object initializer" is consistent with the terminology used by C++.

The syntax for an object using an object initializer is:

```
var obj = { property_1:   value_1,   // property_# may be an identifier...
 // or a string

```

where `obj` is the name of the new object, each `property_i` is an identifier (either a name, a number, or a string literal), and each `value_i` is an expression whose value is assigned to the `property_i`. The `obj` and assignment is optional; if you do not need to refer to this object elsewhere, you do not need to assign it to a variable. (Note that you may need to wrap the object literal in parentheses if the object appears where a statement is expected, so as not to have the literal be confused with a block statement.)

If an object is created with an object initializer in a top-level script, JavaScript interprets the object each time it evaluates an expression containing the object literal. In addition, an initializer used in a function is created each time the function is called.

The following statement creates an object and assigns it to the variable `x` if and only if the expression `cond` is true:

```
if (cond) var x = {hi: "there"};
```

The following example creates `myHonda` with three properties. Note that the `engine` property is also an object with its own properties.

```
var myHonda = {color: "red", wheels: 4, engine: {cylinders: 4, size: 2.2}};
```

You can also use object initializers to create arrays. See array literals.

In JavaScript 1.1 and earlier, you cannot use object initializers. You can create objects only using their constructor functions or using a function supplied by some other object for that purpose. See Using a constructor function.

Using a constructor function

Alternatively, you can create an object with these two steps:

1. Define the object type by writing a constructor function. There is a strong convention, with good reason, to use a capital initial letter.
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, properties, and methods. For example, 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;
}
```

Notice the use of `this` to assign values to the object's properties based on the values passed to the function.

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);
var vpgscar = new Car("Mazda", "Miata", 1990);
```

An object can have a property that is itself another object. For example, 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 McKinnon", 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);
```

Notice that 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 arguments for the owners. Then if you want to find out the name of the owner of car2, you can access the following property:

```
car2.owner.name
```

Note that 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 have to add the property to the definition of the `car` object type.

Using the Object.create method

Objects can also be created using the `Object.create` method. This method can be very useful, because it allows you to choose the prototype object for the object you want to create, without having to define a constructor function. For more detailed information on the method and how to use it, see Object.create method.

Inheritance

All objects in JavaScript inherit from at least one other object. The object being inherited from is known as the prototype, and the inherited properties can be found in the `prototype` object of the constructor.

Indexing object properties

In JavaScript 1.0, you can refer to a property of an object either by its property name or by its ordinal index. In JavaScript 1.1 and later, however, if you initially define a property by its name, you must always refer to it by its name, and if you initially define a property by an index, you must always refer to it by its index.

This restriction applies when you create an object and its properties with a constructor function (as we did previously with the `Car` object type) and when you define individual properties explicitly (for example, `myCar.color = "red"`). If you initially define an object property with an index, such as `myCar[5] = "25 mpg"`, you can subsequently refer to the property only as `myCar[5]`.

The exception to this rule is objects reflected from HTML, such as the `forms` array. You can always refer to objects in these arrays by either their ordinal number (based on where they appear in the document) or their name (if defined). For example, if the second `<FORM>` tag in a document has a `NAME` attribute of "myForm", you can refer to the form as `document.forms[1]` or `document.forms["myForm"]` or `document.myForm`.

Defining properties for an object type

You can add a property to a previously defined object type by using the `prototype` property. This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object. The following code adds a `color` property to all objects of type `car`, and then assigns a value to the `color` property of the object `car1`.

```
Car.prototype.color = null;
car1.color = "black";
```

See the `prototype` property of the `Function` object in the JavaScript Reference for more information.

Defining methods

A *method* is a function associated with an object, or, simply put, a method is a property of an object that is a function. Methods are defined the way normal functions are defined, except that they have to be assigned as the property of an object. Examples are:

```
objectName.methodname = function_name;


```

```
var myObj = {
  myMethod: function(params) {
    // ...do something  }
};
```

where `objectName` is an existing object, `methodname` is the name you are assigning to the method, and `function_name` is the name of the function.

You can then call the method in the context of the object as follows:

```
object.methodname(params);
```

You can define methods for an object type by including a method definition in the object constructor function. For example, you could define a function that would format and display the properties of the previously-defined `car` objects; for example,

```
function displayCar() {
  var result = "A Beautiful " + this.year + " " + this.make
    + " " + this.model;
  pretty_print(result);
}
```

where `pretty_print` is a function to display a horizontal rule and a string. Notice the use of `this` to refer to the object to which the method belongs.

You can make this function a method of `car` by adding the statement

```
this.displayCar = displayCar;
```

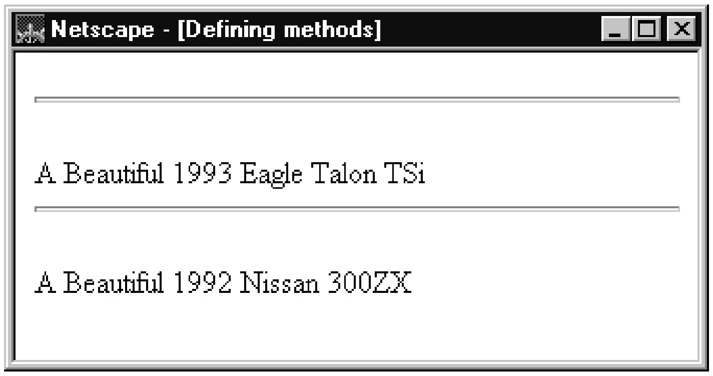
to the object definition. So, the full definition of `car` would now look like

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

Then you can call the `displayCar` method for each of the objects as follows:

```
car1.displayCar();
car2.displayCar();
```

This produces the output shown in the following figure.



A Beautiful 1993 Eagle Talon TSi

A Beautiful 1992 Nissan 300ZX

Figure 7.1: Displaying method output.

Using `this` for object references

JavaScript has a special keyword, `this`, that you can use within a method to refer to the current object. For example, suppose you have a function called `validate` that validates an object's `value` property, given the object and the high and low values:

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

Then, you could call `validate` in each form element's `onchange` event handler, using `this` to pass it the element, as in the following example:

```
<input type="text" name="age" size="3"
  onChange="validate(this, 18, 99)">
```

In general, `this` refers to the calling object in a method.

When combined with the `form` property, `this` can refer to the current object's parent form. In the following example, the form `myForm` contains a `Text` object and a button. When the user clicks the button, the value of the `Text` object is set to the form's name. The button's `onclick` event handler uses `this.form` to refer to the parent form, `myForm`.

```
<form name="myForm">
<p><label>Form name:<input type="text" name="text1" value="Beluga"></label>
   <input name="button1" type="button" value="Show Form Name"
<p>     onclick="this.form.text1.value = this.form.name">
</p>
</form>
```

Defining getters and setters

A *getter* is a method that gets the value of a specific property. A *setter* is a method that sets the value of a specific property. You can define getters and setters on any predefined core object or user-defined object that supports the addition of new properties. The syntax for defining getters and setters uses the object literal syntax.

JavaScript 1.8.1 note

Starting in JavaScript 1.8.1, setters are no longer called when setting properties in object and array initializers.

The following JS shell session illustrates how getters and setters could work for a user-defined object `o`. The JS shell is an application that allows developers to test JavaScript code in batch mode or interactively. In Firefox you can get a JS shell by pressing Ctrl+Shift+K.

```
js> var o = {a: 7, get b() {return this.a + 1;}, set c(x) {this.a = x / 2}};
[object Object]
js> o.a;
7
js> o.b;
8
js> o.c = 50;
js> o.a;
25
```

The `o` object's properties are:

- `o.a` — a number

- o.b — a getter that returns o.a plus 1
- o.c — a setter that sets the value of o.a to half of the value o.c is being set to

Please note that function names of getters and setters defined in an object literal using "[gs]et *property*()" (as opposed to __define[GS]etter__) are not the names of the getters themselves, even though the [gs]et *propertyName*(){ } syntax may mislead you to think otherwise. To name a function in a getter or setter using the "[gs]et *property*()" syntax, define an explicitly named function programmatically using Object.defineProperty (or the Object.prototype.__defineGetter__ legacy fallback).

This JavaScript shell session illustrates how getters and setters can extend the Date prototype to add a year property to all instances of the predefined Date class. It uses the Date class's existing getFullYear and setFullYear methods to support the year property's getter and setter.

These statements define a getter and setter for the year property:

```
js> var d = Date.prototype;
js> Object.defineProperty(d, "year", {
    get: function() {return this.getFullYear() },
    set: function(y) { this.setFullYear(y) }
});
```

These statements use the getter and setter in a Date object:

```
js> var now = new Date;
js> print(now.year);
2000
js> now.year = 2001;
987617605170
js> print(now);
Wed Apr 18 11:13:25 GMT-0700 (Pacific Daylight Time) 2001
```

Obsolete syntaxes

In the past, JavaScript supported several other syntaxes for defining getters and setters. None of these syntaxes were supported by other engines, and support has been removed in recent versions of JavaScript. See this dissection of the removed syntaxes for further details on what was removed and how to adapt to those removals.

Summary

In principle, getters and setters can be either

- defined using object initializers, or
- added later to any object at any time using a getter or setter adding method.

When defining getters and setters using object initializers all you need to do is to prefix a getter method with `get` and a setter method with `set`. Of course, the getter method must not expect a parameter, while the setter method expects exactly one parameter (the new value to set). For instance:

```
var o = {
  a: 7,
  get b() { return this.a + 1; },
  set c(x) { this.a = x / 2; }
};
```

Getters and setters can also be added to an object at any time after creation using the `Object.defineProperties` method. This method's first parameter is the object on which you want to define the getter or setter. The second parameter is an object whose property names are the getter or setter names, and whose property values are objects for defining the getter or setter functions. Here's an example that defines the same getter and setter used in the previous example:

```
var o = { a:0 }

Object.defineProperties(o, {
    "b": { get: function () { return this.a + 1; } },
    "c": { set: function (x) { this.a = x / 2; } }
});

o.c = 10 // Runs the setter, which assigns 10 / 2 (5) to the 'a' propertyconsole.lo
 // Runs the getter, which yields a + 1 or 6
```

Which of the two forms to choose depends on your programming style and task at hand. If you already go for the object initializer when defining a prototype you will probably most of the time choose the first form. This form is more compact and natural. However, if you need to add getters and setters later — because you did not write the prototype or particular object — then the second form is the only possible form. The second form probably best represents the dynamic nature of JavaScript — but it can make the code hard to read and understand.

Prior to Firefox 3.0, getter and setter are not supported for DOM Elements. Older versions of Firefox silently fail. If exceptions are needed for those, changing the prototype of HTMLElement (`HTMLElement.prototype.__define[SG]etter__`) and throwing an exception is a workaround.

With Firefox 3.0, defining getter or setter on an already-defined property will throw an exception. The property must be deleted beforehand, which is not the case for older versions of Firefox.

See also

- `Object.defineProperty`
- `get`
- `set`

Deleting properties

You can remove a non-inherited property by using the `delete` operator. The following code shows how to remove a property.

```
//Creates a new object, myobj, with two properties, a and b.var myobj = new Objec
myobj.a = 5;
myobj.b = 12;

//Removes the a property, leaving myobj with only the b property.delete myobj.a;
console.log ("a" in myobj) // yields "false"

```

You can also use `delete` to delete a global variable if the `var` keyword was not used to declare the variable:

```
g = 17;
delete g;
```

See `delete` for more information.

See also

- ECMAScript 5.1 spec: Language Overview
- JavaScript. The core. (Dmitry A. Soshnikov ECMA-262 article series)

« Previous Next »

Predefined Core Objects

This chapter describes the predefined objects in core JavaScript: `Array`, `Boolean`, `Date`, `Function`, `Math`, `Number`, `RegExp`, and `String`.

Array Object

JavaScript does not have an explicit array data type. However, you can use the predefined `Array` object and its methods to work with arrays in your applications. The `Array` object has methods for manipulating arrays in various ways, such as joining, reversing, and sorting them. It has a property for determining the array length and other properties for use with regular expressions.

An *array* is an ordered set of values that you refer to with a name and an index. For example, you could have an array called `emp` that contains employees' names indexed by their employee number. So `emp[1]` would be employee number one, `emp[2]` employee number two, and so on.

Creating an Array

The following statements create equivalent arrays:

```
var arr = new Array(element0, element1, ..., elem
var arr = Array(element0, element1, ..., elementN
var arr = [element0, element1, ..., elementN];
```

`element0, element1, ..., elementN` is a list of values for the array's elements. When these values are specified, the array is initialized with them as the array's elements. The array's `length` property is set to the number of arguments.

The bracket syntax is called an "array literal" or "array initializer." It's shorter than other forms of array creation, and so is generally preferred. See Array Literals for details.

To create an Array with non-zero length, but without any items, either of the following can be used:

```
var arr = new Array(arrayLength);
var arr = Array(arrayLength);

// This has exactly the same effectvar arr = [];
arr.length = arrayLength;

```

Note: in the above code, `arrayLength` must be a `Number`. Otherwise, an array with a single element (the provided value) will be created. Calling `arr.length` will return `arrayLength`, but the array actually contains empty (undefined) elements. Running a for...in loop on the array will return none of the array's elements.

In addition to a newly defined variable as shown above, Arrays can also be assigned as a property of a new or an existing object:

```

var obj = {};// ...obj.prop = [element0, element1, ..., elementN];

// ORvar obj = {prop: [element0, element1, ...., elementN]}

```

If you wish to initialize an array with a single element, and the element happens to be a `Number`, you must use the bracket syntax. When a single `Number` value is passed to the Array() constructor or function, it is interpreted as an `arrayLength`, not as a single element.

```
var arr = [42];
var arr = Array(42); // Creates an array with no element, but with arr.length set t

// The above code is equivalent to
var arr = [];
arr.length = 42;

```

Calling `Array(N)` results in a `RangeError`, if `N` is a non-whole number whose fractional portion is non-zero. The following example illustrates this behavior.

```
var arr = Array(9.3);  // RangeError: Invalid array length
```

If your code needs to create arrays with single elements of an arbitrary data type, it is safer to use array literals. Or, create an empty array first before adding the single element to it.

Populating an Array

You can populate an array by assigning values to its elements. For example,

```
var emp = [];
emp[0] = "Casey Jones";
emp[1] = "Phil Lesh";
emp[2] = "August West";
```

Note: if you supply a non-integer value to the array operator in the code above, a property will be created in the object representing the array, instead of an array element.

```
 var arr = [];
arr[3.4] = "Oranges";
console.log(arr.length);                // 0
console.log(arr.hasOwnProperty(3.4));   // true
```

You can also populate an array when you create it:

```
var myArray = new Array("Hello", myVar, 3.14159);
var myArray = ["Mango", "Apple", "Orange"]
```

Referring to Array Elements

You refer to an array's elements by using the element's ordinal number. For example, suppose you define the following array:

```
var myArray = ["Wind", "Rain", "Fire"];
```

You then refer to the first element of the array as `myArray[0]` and the second element of the array as `myArray[1]`. The index of the elements begins with zero.

Note: the array operator (square brackets) is also used for accessing the array's properties (arrays are also objects in JavaScript). For example,

```
 var arr = ["one", "two", "three"];
arr[2];  // three
```

```
arr["length"];  // 3
```

Understanding length

At the implementation level, JavaScript's arrays actually store their elements as standard object properties, using the array index as the property name. The `length` property is special; it always returns the index of the last element plus one (in following example Dusty is indexed at 30 so cats.length returns 30 + 1). Remember, Javascript Array indexes are 0-based: they start at 0, not 1. This means that the `length` property will be one more than the highest index stored in the array:

```
var cats = [];
cats[30] = ['Dusty'];
print(cats.length); // 31
```

You can also assign to the `length` property. Writing a value that is shorter than the number of stored items truncates the array; writing 0 empties it entirely:

```
var cats = ['Dusty', 'Misty', 'Twiggy'];
console.log(cats.length); // 3
cats.length = 2;
console.log(cats); // prints "Dusty,Misty" - Twiggy has been removed
cats.length = 0;
console.log(cats); // prints nothing; the cats array is empty
cats.length = 3;
console.log(cats); // [undefined, undefined, undefined]

```

Iterating over arrays

A common operation is to iterate over the values of an array, processing each one in some way. The simplest way to do this is as follows:

```
var colors = ['red', 'green', 'blue'];
for (var i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}
```

If you know that none of the elements in your array evaluate to `false` in a boolean context — if your array consists only of DOM nodes, for example, you can use a more efficient idiom:

```
var divs = document.getElementsByTagName('div');
for (var i = 0, div; div = divs[i]; i++) {
  /* Process div in some way */
}
```

This avoids the overhead of checking the length of the array, and ensures that the `div` variable is reassigned to the current item each time around the loop for added convenience.

Introduced in JavaScript 1.6

The `forEach()` method, introduced in JavaScript 1.6, provides another way of iterating over an array:

```
var colors = ['red', 'green', 'blue'];
colors.forEach(function(color) {
  console.log(color);
});
```

The function passed to `forEach` is executed once for every item in the array, with the array item passed as the argument to the function. Unassigned values are not iterated in a `forEach` loop.

Note that the elements of array omitted when the array is defined are not listed when iterating by `forEach`, but are listed when `undefined` has been manually assigned to the element:

```
var array = ['first', 'second', , 'fourth'];

// returns ['first', 'second', 'fourth'];array.forEach(function(element) {
  console.log(element);
})

if(array[2] === undefined) { console.log('array[2] is undefined'); } // true
var array = ['first', 'second', undefined, 'fourth'];

// returns ['first', 'second', undefined, 'fourth'];array.forEach(function(elemer
  console.log(element);
})

```

Since JavaScript elements are saved as standard object properties, it is not advisable to iterate through JavaScript arrays using for...in loops because normal elements and all enumerable properties will be listed.

Array Methods

The `Array` object has the following methods:

- `concat()` joins two arrays and returns a new array.

```
var myArray = new Array("1", "2", "3");
myArray = myArray.concat("a", "b", "c");
  // myArray is now ["1", "2", "3", "a", "b", "c"]
```

- `join(deliminator = ",")` joins all elements of an array into a string.

```
var myArray = new Array("Wind", "Rain", "Fire");
var list = myArray.join(" - "); // list is "Wind - Rain - Fire"
```

- `push()` adds one or more elements to the end of an array and returns the resulting length of the array.

```
var myArray = new Array("1", "2");
myArray.push("3"); // myArray is now ["1", "2", "3"]
```

- `pop()` removes the last element from an array and returns that element.

```
var myArray = new Array("1", "2", "3");
var last = myArray.pop(); // myArray is now ["1", "2"], last = "3"
```

- `shift()` removes the first element from an array and returns that element.

```
var myArray = new Array ("1", "2", "3");
var first = myArray.shift(); // myArray is now ["2", "3"], first is "1"
```

- `unshift()` adds one or more elements to the front of an array and returns the new length of the array.

```
var myArray = new Array ("1", "2", "3");
myArray.unshift("4", "5"); // myArray becomes ["4", "5", "1", "2", "3"]
```

- `slice(start_index, upto_index)` extracts a section of an array and returns a new array.

```
var myArray = new Array ("a", "b", "c", "d", "e");
                                /* starts at index 1 and extracts all elements
myArray = myArray.slice(1, 4);    until index 3, returning [ "b", "c", "d"] */
```

- `splice(index, count_to_remove, addelement1, addelement2, ...)` removes elements from an array and (optionally) replaces them.

```
var myArray = new Array ("1", "2", "3", "4", "5");
myArray.splice(1, 3, "a", "b", "c", "d"); // myArray is now ["1", "a", "b",
 // and then inserted all consecutive elements in its place.

```

- `reverse()` transposes the elements of an array: the first array element becomes the last and the last becomes the first.

```
var myArray = new Array ("1", "2", "3");
myArray.reverse();
 // transposes the array so that myArray = [ "3", "2", "1" ]
```

- `sort()` sorts the elements of an array.

```
var myArray = new Array("Wind", "Rain", "Fire");
myArray.sort();
 // sorts the array so that myArrray = [ "Fire", "Rain", "Wind" ]
```

`sort()` can also take a callback function to determine how array elements are compared. The function compares two values and returns one of three values:

  - if `a` is less than `b` by the sorting system, return -1 (or any negative number)
  - if `a` is greater than `b` by the sorting system, return 1 (or any positive number)
  - if `a` and `b` are considered equivalent, return 0.

For instance, the following will sort by the last letter of an array:

```
var sortFn = function(a, b){
  if (a[a.length - 1] < b[b.length - 1]) return -1;
  if (a[a.length - 1] > b[b.length - 1]) return 1;
  if (a[a.length - 1] == b[b.length - 1]) return 0;
}
```

```
myArray.sort(sortFn);
 // sorts the array so that myArray = ["Wind","Fire","Rain"]
```

Introduced in JavaScript 1.6

Compatibility code for older browsers can be found for each of these functions on the individual pages. Native browser support for these features in various browsers can be found here.

- `indexOf(searchElement[, fromIndex])` searches the array for `searchElement` and returns the index of the first match.

```
var a = ['a', 'b', 'a', 'b', 'a'];
alert(a.indexOf('b')); // Alerts 1// Now try again, starting from after the
 // Alerts -1, because 'z' was not found

```

- `lastIndexOf(searchElement[, fromIndex])` works like `indexOf`, but starts at the end and searches backwards.

```
var a = ['a', 'b', 'c', 'd', 'a', 'b'];
alert(a.lastIndexOf('b')); // Alerts 5// Now try again, starting from before
 // Alerts -1

```

- `forEach(callback[, thisObject])` executes `callback` on every array item.

```
var a = ['a', 'b', 'c'];
a.forEach(alert); // Alerts each item in turn
```

- `map(callback[, thisObject])` returns a new array of the return value from executing `callback` on every array item.

```
var a1 = ['a', 'b', 'c'];
var a2 = a1.map(function(item) { return item.toUpperCase(); });
alert(a2); // Alerts A,B,C
```

- `filter(callback[, thisObject])` returns a new array containing the items for which callback returned true.

```
var a1 = ['a', 10, 'b', 20, 'c', 30];
var a2 = a1.filter(function(item) { return typeof item == 'number'; });
alert(a2); // Alerts 10,20,30
```

- `every(callback[, thisObject])` returns true if `callback` returns true for every item in the array.

```
function isNumber(value){
  return typeof value == 'number';
}
var a1 = [1, 2, 3];
alert(a1.every(isNumber)); // Alerts truevar a2 = [1, '2', 3];
alert(a2.every(isNumber)); // Alerts false
```

- `some(callback[, thisObject])` returns true if `callback` returns true for at least one item in the array.

```
function isNumber(value){
  return typeof value == 'number';
}
var a1 = [1, 2, 3];
alert(a1.some(isNumber)); // Alerts truevar a2 = [1, '2', 3];
alert(a2.some(isNumber)); // Alerts truevar a3 = ['1', '2', '3'];
alert(a3.some(isNumber)); // Alerts false
```

The methods above that take a callback are known as *iterative methods*, because they iterate over the entire array in some fashion. Each one takes an optional second argument called `thisObject`. If provided, `thisObject` becomes the value of the `this` keyword inside the body of the callback function. If not provided, as with other cases where a function is invoked outside of an explicit object context, `this` will refer to the global object (`window`).

The callback function is actually called with three arguments. The first is the value of the current item, the second is its array index, and the third is a reference to the array itself. JavaScript functions ignore any arguments that are not named in the parameter list so it is safe to provide a callback function that only takes a single argument, such as `alert`.

Introduced in JavaScript 1.8

- `reduce(callback[, initialValue])` applies `callback(firstValue, secondValue)` to reduce the list of items down to a single value.

  ```
  var a = [10, 20, 30];
  var total = a.reduce(function(first, second) { return first + second; }, 0);
  alert(total) // Alerts 60
  ```

- `reduceRight(callback[, initialValue])` works like `reduce()`, but starts with the last element.

`reduce` and `reduceRight` are the least obvious of the iterative array methods. They should be used for algorithms that combine two values recursively in order to reduce a sequence down to a single value.

Multi-Dimensional Arrays

Arrays can be nested, meaning that an array can contain another array as an element. Using this characteristic of JavaScript arrays, multi-dimensional arrays can be created.

The following code creates a two-dimensional array.

```
var a = new Array(4);
for (i = 0; i < 4; i++) {
  a[i] = new Array(4);
  for (j = 0; j < 4; j++) {
    a[i][j] = "[" + i + "," + j + "]";
  }
}
```

This example creates an array with the following rows:

```
Row 0: [0,0] [0,1] [0,2] [0,3]
Row 1: [1,0] [1,1] [1,2] [1,3]
Row 2: [2,0] [2,1] [2,2] [2,3]
Row 3: [3,0] [3,1] [3,2] [3,3]
```

Arrays and Regular Expressions

When an array is the result of a match between a regular expression and a string, the array returns properties and elements that provide information about the match. An array is the return value of `RegExp.exec()`, `String.match()`, and `String.split()`. For information on using arrays with regular expressions, see Regular Expressions.

Working with Array-like objects

Introduced in JavaScript 1.6

Some JavaScript objects, such as the `NodeList` returned by `document.getElementsByTagName()` or the `arguments` object made available within the body of a function, look and behave like arrays on the surface but do not share all of their methods. The `arguments` object provides a `length` attribute but does not implement the `forEach()` method, for example.

Array generics, introduced in JavaScript 1.6, provide a way of running `Array` methods against other array-like objects. Each standard array method has a corresponding method on the `Array` object itself; for example:

```
function alertArguments() {
   Array.forEach(arguments, function(item) {
     alert(item);
   });
 }
```

These generic methods can be emulated more verbosely in older versions of JavaScript using the call method provided by JavaScript function objects:

```
Array.prototype.forEach.call(arguments, function(item) {
   alert(item);
 });
```

Array generic methods can be used on strings as well, since they provide sequential access to their characters in a similar way to arrays:

```
Array.forEach("a string", function(chr) {
   alert(chr);
});
```

Here are some further examples of applying array methods to strings, also taking advantage of JavaScript 1.8 expression closures:

```
var str = 'abcdef';
var consonantsOnlyStr = Array.filter(str, function (c) !(/[aeiou]/i).test(c)).joi

// 21 (reduce() since JS v1.8)

```

Note that `filter` and `map` do not automatically return the characters back into being members of a string in the return result; an array is returned, so we must use `join` to return back to a string.

Array comprehensions

Introduced in JavaScript 1.7

Introduced in JavaScript 1.7, array comprehensions provide a useful shortcut for constructing a new array based on the contents of another. Comprehensions can often be used in place of calls to `map()` and `filter()`, or as a way of combining the two.

The following comprehension takes an array of numbers and creates a new array of the double of each of those numbers.

```
var numbers = [1, 2, 3, 4];
var doubled = [i * 2 for (i of numbers)];
alert(doubled); // Alerts 2,4,6,8
```

This is equivalent to the following `map()` operation:

```
var doubled = numbers.map(function(i){return i * 2;});
```

Comprehensions can also be used to select items that match a particular expression. Here is a comprehension which selects only even numbers:

```
var numbers = [1, 2, 3, 21, 22, 30];
var evens = [i for (i of numbers) if (i % 2 === 0)];
alert(evens); // Alerts 2,22,30
```

`filter()` can be used for the same purpose:

```
var evens = numbers.filter(function(i){return i % 2 === 0;});
```

`map()` and `filter()` style operations can be combined into a single array comprehension. Here is one that filters just the even numbers, then creates an array containing their doubles:

```
var numbers = [1, 2, 3, 21, 22, 30];
var doubledEvens = [i * 2 for (i of numbers) if (i % 2 === 0)];
alert(doubledEvens); // Alerts 4,44,60
```

The square brackets of an array comprehension introduce an implicit block for scoping purposes. New variables (such as i in the example) are treated as if they had been declared using `let`. This means that they will not be available outside of the comprehension.

The input to an array comprehension does not itself need to be an array; iterators and generators can also be used.

Even strings may be used as input; to achieve the filter and map actions (under Array-like objects) above:

```
var str = 'abcdef';
var consonantsOnlyStr = [c for (c of str) if (!(/[aeiouAEIOU]/).test(c))  ].join('
 // 'a0b0c0d0e0f0'
```

Again, the input form is not preserved, so we have to use `join()` to revert back to a string.

Boolean Object

The `Boolean` object is a wrapper around the primitive Boolean data type. Use the following syntax to create a `Boolean` object:

```
var booleanObjectName = new Boolean(value);
```

Do not confuse the primitive Boolean values `true` and `false` with the true and false values of the `Boolean` object. Any object whose value is not `undefined` , `null`, `0`, `NaN`, or the empty string, including a `Boolean` object whose value is false, evaluates to true when passed to a conditional statement. See if...else Statement for more information.

Date Object

JavaScript does not have a date data type. However, you can use the `Date` object and its methods to work with dates and times in your applications. The `Date` object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties.

JavaScript handles dates similarly to Java. The two languages have many of the same date methods, and both languages store dates as the number of milliseconds since January 1, 1970, 00:00:00.

The `Date` object range is -100,000,000 days to 100,000,000 days relative to 01 January, 1970 UTC.

To create a `Date` object:

```
var dateObjectName = new Date([parameters]);
```

where `dateObjectName` is the name of the `Date` object being created; it can be a new object or a property of an existing object.

Calling Date without the new keyword simply converts the provided date to a string representation.

The `parameters` in the preceding syntax can be any of the following:

- Nothing: creates today's date and time. For example, `today = new Date();`.
- A string representing a date in the following form: "Month day, year hours:minutes:seconds." For example, `var Xmas95 = new Date("December 25, 1995 13:30:00")`. If you omit hours, minutes, or seconds, the value will be set to zero.
- A set of integer values for year, month, and day. For example, `var Xmas95 = new Date(1995, 11, 25)`.
- A set of integer values for year, month, day, hour, minute, and seconds. For example, `var Xmas95 = new Date(1995, 11, 25, 9, 30, 0);`.

JavaScript 1.2 and earlier

The `Date` object behaves as follows:

- Dates prior to 1970 are not allowed.
- JavaScript depends on platform-specific date facilities and behavior; the behavior of the `Date` object varies from platform to platform.

Methods of the Date Object

The `Date` object methods for handling dates and times fall into these broad categories:

- "set" methods, for setting date and time values in `Date` objects.
- "get" methods, for getting date and time values from `Date` objects.
- "to" methods, for returning string values from `Date` objects.
- parse and UTC methods, for parsing `Date` strings.

With the "get" and "set" methods you can get and set seconds, minutes, hours, day of the month, day of the week, months, and years separately. There is a `getDay` method that returns the day of the week, but no corresponding `setDay` method, because the day of the week is set automatically. These methods use integers to represent these values as follows:

- Seconds and minutes: 0 to 59
- Hours: 0 to 23
- Day: 0 (Sunday) to 6 (Saturday)
- Date: 1 to 31 (day of the month)
- Months: 0 (January) to 11 (December)
- Year: years since 1900

For example, suppose you define the following date:

```
var Xmas95 = new Date("December 25, 1995");
```

Then `Xmas95.getMonth()` returns 11, and `Xmas95.getFullYear()` returns 1995.

The `getTime` and `setTime` methods are useful for comparing dates. The `getTime` method returns the number of milliseconds since January 1, 1970, 00:00:00 for a `Date` object.

For example, the following code displays the number of days left in the current year:

```
var today = new Date();
var endYear = new Date(1995, 11, 31, 23, 59, 59, 999); // Set day and monthendYea
var daysLeft = Math.round(daysLeft); //returns days left in the year

```

This example creates a `Date` object named `today` that contains today's date. It then creates a `Date` object named `endYear` and sets the year to the current year. Then, using the number of milliseconds per day, it computes the number of days between `today` and `endYear`, using `getTime` and rounding to a whole number of days.

The `parse` method is useful for assigning values from date strings to existing `Date` objects. For example, the following code uses `parse` and `setTime` to assign a date value to the `IPOdate` object:

```
var IPOdate = new Date();
IPOdate.setTime(Date.parse("Aug 9, 1995"));
```

Using the Date Object: an Example

In the following example, the function `JSClock()` returns the time in the format of a digital clock.

```
function JSClock() {
  var time = new Date();
  var hour = time.getHours();
  var minute = time.getMinutes();
  var second = time.getSeconds();
  var temp = "" + ((hour > 12) ? hour - 12 : hour);
  if (hour == 0)
    temp = "12";
  temp += ((minute < 10) ? ":0" : ":") + minute;
  temp += ((second < 10) ? ":0" : ":") + second;
  temp += (hour >= 12) ? " P.M." : " A.M.";
  return temp;
}
```

The `JSClock` function first creates a new `Date` object called `time`; since no arguments are given, time is created with the current date and time. Then calls to the `getHours`, `getMinutes`, and `getSeconds` methods assign the value of the current hour, minute, and second to `hour`, `minute`, and `second`.

The next four statements build a string value based on the time. The first statement creates a variable `temp`, assigning it a value using a conditional expression; if `hour` is greater than 12, (`hour - 12`), otherwise simply hour, unless hour is 0, in which case it becomes 12.

The next statement appends a `minute` value to `temp`. If the value of `minute` is less than 10, the conditional expression adds a string with a preceding zero; otherwise it adds a string with a demarcating colon. Then a statement appends a seconds value to `temp` in the same way.

Finally, a conditional expression appends "PM" to `temp` if `hour` is 12 or greater; otherwise, it appends "AM" to `temp`.

Function Object

The predefined `Function` object specifies a string of JavaScript code to be compiled as a function.

To create a `Function` object:

```
var functionObjectName = new Function ([arg1, arg2, ... argn], functionBody);
```

`functionObjectName` is the name of a variable or a property of an existing object. It can also be an object followed by a lowercase event handler name, such as `window.onerror`.

`arg1`, `arg2`, ... `argn` are arguments to be used by the function as formal argument names. Each must be a string that corresponds to a valid JavaScript identifier; for example "x" or "theForm".

`functionBody` is a string specifying the JavaScript code to be compiled as the function body.

`Function` objects are evaluated each time they are used. This is less efficient than declaring a function and calling it within your code, because declared functions are compiled.

In addition to defining functions as described here, you can also use the `function` statement and the function expression. See the JavaScript Reference for more information.

The following code assigns a function to the variable `setBGColor`. This function sets the current document's background color.

```
var setBGColor = new Function("document.bgColor = 'antiquewhite'");
```

To call the `Function` object, you can specify the variable name as if it were a function. The following code executes the function specified by the `setBGColor` variable:

```
var colorChoice="antiquewhite";
if (colorChoice=="antiquewhite") {setBGColor()}
```

You can assign the function to an event handler in either of the following ways:

```
document.form1.colorButton.onclick = setBGColor;
```

```
<INPUT NAME="colorButton" TYPE="button"
  VALUE="Change background color"
  onClick="setBGColor()">
```

Creating the variable `setBGColor` shown above is similar to declaring the following function:

```
function setBGColor() {
  document.bgColor = 'antiquewhite';
}
```

Assigning a function to a variable is similar to declaring a function, but there are differences:

- When you assign a function to a variable using `var setBGColor = new Function("...")`, `setBGColor` is a variable for which the current value is a reference to the function created with `new Function()`.
- When you create a function using `function setBGColor() {...}`, `setBGColor` is not a variable, it is the name of a function.

You can nest a function within a function. The nested (inner) function is private to its containing (outer) function:

- The inner function can be accessed only from statements in the outer function.
- The inner function can use the arguments and variables of the outer function. The outer function cannot use the arguments and variables of the inner function.

Math Object

The predefined `Math` object has properties and methods for mathematical constants and functions. For example, the `Math` object's `PI` property has the value of pi (3.141...), which you would use in an application as

```
Math.PI
```

Similarly, standard mathematical functions are methods of `Math`. These include trigonometric, logarithmic, exponential, and other functions. For example, if you want to use the trigonometric function sine, you would write

```
Math.sin(1.56)
```

Note that all trigonometric methods of `Math` take arguments in radians.

The following table summarizes the `Math` object's methods.

Table 7.1 Methods of Math

Method	Description
`abs`	Absolute value
`sin`, `cos`, `tan`	Standard trigonometric functions; argument in radians
`acos`, `asin`, `atan`, `atan2`	Inverse trigonometric functions; return values in radians
`exp`, `log`	Exponential and natural logarithm, base `e`
`ceil`	Returns least integer greater than or equal to argument
`floor`	Returns greatest integer less than or equal to argument
`min`, `max`	Returns greater or lesser (respectively) of two arguments
`pow`	Exponential; first argument is base, second is exponent
`random`	Returns a random number between 0 and 1.
`round`	Rounds argument to nearest integer
`sqrt`	Square root

Unlike many other objects, you never create a `Math` object of your own. You always use the predefined `Math` object.

Number Object

The `Number` object has properties for numerical constants, such as maximum value, not-a-number, and infinity. You cannot change the values of these properties and you use them as follows:

```
var biggestNum = Number.MAX_VALUE;
var smallestNum = Number.MIN_VALUE;
var infiniteNum = Number.POSITIVE_INFINITY;
var negInfiniteNum = Number.NEGATIVE_INFINITY;
var notANum = Number.NaN;
```

You always refer to a property of the predefined `Number` object as shown above, and not as a property of a `Number` object you create yourself.

The following table summarizes the `Number` object's properties.

Table 7.2 Properties of Number

Property	Description
`MAX_VALUE`	The largest representable number
`MIN_VALUE`	The smallest representable number
`NaN`	Special "not a number" value
`NEGATIVE_INFINITY`	Special negative infinite value; returned on overflow
`POSITIVE_INFINITY`	Special positive infinite value; returned on overflow

The Number prototype provides methods for retrieving information from Number objects in various formats. The following table summarizes the methods of `Number.prototype`.

Table 7.3 Methods of Number.prototype

Method	Description
`toExponential`	Returns a string representing the number in exponential notation.
`toFixed`	Returns a string representing the number in fixed-point notation.
`toPrecision`	Returns a string representing the number to a specified precision in fixed-point notation.
`toSource`	Returns an object literal representing the specified `Number` object; you can use this value to create a new object. Overrides the `Object.toSource` method.
`toString`	Returns a string representing the specified object. Overrides the `Object.toString` method.
`valueOf`	Returns the primitive value of the specified object. Overrides the

	`Object.valueOf` method.

RegExp Object

The `RegExp` object lets you work with regular expressions. It is described in Regular Expressions.

String Object

The `String` object is a wrapper around the string primitive data type. Do not confuse a string literal with the `String` object. For example, the following code creates the string literal `s1` and also the `String` object `s2`:

```
var s1 = "foo"; //creates a string literal valuevar s2 = new String("foo");
 //creates a String object
```

You can call any of the methods of the `String` object on a string literal value—JavaScript automatically converts the string literal to a temporary `String` object, calls the method, then discards the temporary `String` object. You can also use the `String.length` property with a string literal.

You should use string literals unless you specifically need to use a `String` object, because `String` objects can have counterintuitive behavior. For example:

```
var s1 = "2 + 2"; //creates a string literal valuevar s2 = new String("2 + 2"); /
 //returns the string "2 + 2"

```

A `String` object has one property, `length`, that indicates the number of characters in the string. For example, the following code assigns `x` the value 13, because "Hello, World!" has 13 characters:

```
var mystring = "Hello, World!";
var x = mystring.length;
```

A `String` object has two types of methods: those that return a variation on the string itself, such as `substring` and `toUpperCase`, and those that return an HTML-formatted version of the string, such as `bold` and `link`.

For example, using the previous example, both `mystring.toUpperCase()` and "`hello,` world!".toUpperCase() return the string "HELLO, WORLD!"

The substring method takes two arguments and returns a subset of the string between the two arguments. Using the previous example, mystring.substring(4, 9) returns the string "o, Wo". See the substring method of the String object in the JavaScript Reference for more information.

The String object also has a number of methods for automatic HTML formatting, such as bold to create boldface text and link to create a hyperlink. For example, you could create a hyperlink to a hypothetical URL with the link method as follows:

```
mystring.link("http://www.helloworld.com")
```

The following table summarizes the methods of String objects.

Table 7.4 Methods of String Instances

Method	Description
anchor	Creates HTML named anchor.
big, blink, bold, fixed, italics, small, strike, sub, sup	Create HTML formatted string.
charAt, charCodeAt	Return the character or character code at the specified position in string.
indexOf, lastIndexOf	Return the position of specified substring in the string or last position of specified substring, respectively.
link	Creates HTML hyperlink.
concat	Combines the text of two strings and returns a new string.
fromCharCode	Constructs a string from the specified sequence of Unicode values. This is a method of the String class, not a String instance.
split	Splits a String object into an array of strings by separating the string into substrings.
slice	Extracts a section of an string and returns a new string.
substring, substr	Return the specified subset of the string, either by specifying the start and end indexes or the start index and a length.
match, replace, search	Work with regular expressions.

`toLowerCase`, `toUpperCase`	Return the string in all lowercase or all uppercase, respectively.

« Previous Next »

Details of the object model

This article is in need of a technical review.

JavaScript is an object-based language based on prototypes, rather than being class-based. Because of this different basis, it can be less apparent how JavaScript allows you to create hierarchies of objects and to have inheritance of properties and their values. This chapter attempts to clarify the situation.

This chapter assumes that you are already somewhat familiar with JavaScript and that you have used JavaScript functions to create simple objects.

Class-based vs. prototype-based languages

Class-based object-oriented languages, such as Java and C++, are founded on the concept of two distinct entities: classes and instances.

- A *class* defines all of the properties (considering methods and fields in Java, or members in C++, to be properties) that characterize a certain set of objects. A class is an abstract thing, rather than any particular member of the set of objects it describes. For example, the `Employee` class could represent the set of all employees.
- An *instance*, on the other hand, is the instantiation of a class; that is, one of its members. For example, `Victoria` could be an instance of the `Employee` class, representing a particular individual as an employee. An instance has exactly the properties of its parent class (no more, no less).

A prototype-based language, such as JavaScript, does not make this distinction: it simply has objects. A prototype-based language has the notion of a *prototypical object*, an object used as a template from which to get the initial properties for a new object. Any object can specify its own properties, either when you create it or at run time. In addition, any object can be associated as the *prototype* for another object, allowing the second object to share the first object's properties.

Defining a class

In class-based languages, you define a class in a separate *class definition*. In that definition you can specify special methods, called *constructors*, to create instances of the class. A constructor method can specify initial values for the instance's properties and perform other processing appropriate at creation time. You use the `new` operator in association with the constructor method to create class instances.

JavaScript follows a similar model, but does not have a class definition separate from the constructor. Instead, you define a constructor function to create objects with a particular initial set of properties and values. Any JavaScript function can be used as a constructor. You use the `new` operator with a constructor function to create a new object.

Subclasses and inheritance

In a class-based language, you create a hierarchy of classes through the class definitions. In a class definition, you can specify that the new class is a *subclass* of an already existing class. The subclass inherits all the properties of the superclass and additionally can add new properties or modify the inherited ones. For example, assume the `Employee` class includes only the `name` and `dept` properties, and `Manager` is a subclass of `Employee` that adds the `reports` property. In this case, an instance of the `Manager` class would have all three properties: `name`, `dept`, and `reports`.

JavaScript implements inheritance by allowing you to associate a prototypical object with any constructor function. So, you can create exactly the `Employee` — `Manager` example, but you use slightly different terminology. First you define the `Employee` constructor function, specifying the `name` and `dept` properties. Next, you define the `Manager` constructor function, specifying the `reports` property. Finally, you assign a new `Employee` object as the `prototype` for the `Manager` constructor function. Then, when you create a new `Manager`, it inherits the `name` and `dept` properties from the `Employee` object.

Adding and removing properties

In class-based languages, you typically create a class at compile time and then you instantiate instances of the class either at compile time or at run time. You cannot change the number or the type of properties of a class after you define the class. In JavaScript, however, at run time you can add or remove properties of any object. If you add a property to an object that is used as the prototype for a set of objects, the objects for which it is the prototype also get the new property.

Summary of differences

The following table gives a short summary of some of these differences. The rest of this chapter describes the details of using JavaScript constructors and prototypes to create an object hierarchy and compares this to how you would do it in Java.

Table 8.1 Comparison of class-based (Java) and prototype-based (JavaScript) object systems

Class-based (Java)	Prototype-based (JavaScript)
Class and instance are distinct entities.	All objects are instances.
Define a class with a class definition; instantiate a class with constructor methods.	Define and create a set of objects with constructor functions.
Create a single object with the `new` operator.	Same.
Construct an object hierarchy by using class definitions to define subclasses of existing classes.	Construct an object hierarchy by assigning an object as the prototype associated with a constructor function.
Inherit properties by following the class chain.	Inherit properties by following the prototype chain.
Class definition specifies *all* properties of all instances of a class. Cannot add properties dynamically at run time.	Constructor function or prototype specifies an *initial set* of properties. Can add or remove properties dynamically to individual objects or to the entire set of objects.

The employee example

The remainder of this chapter uses the employee hierarchy shown in the following figure.

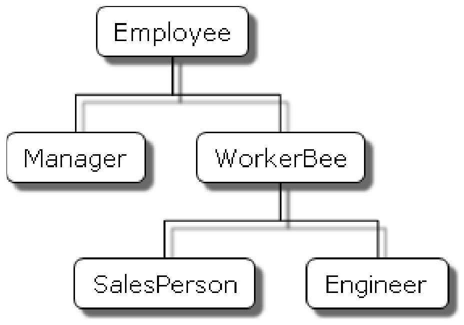


Figure 8.1: A simple object hierarchy

This example uses the following objects:

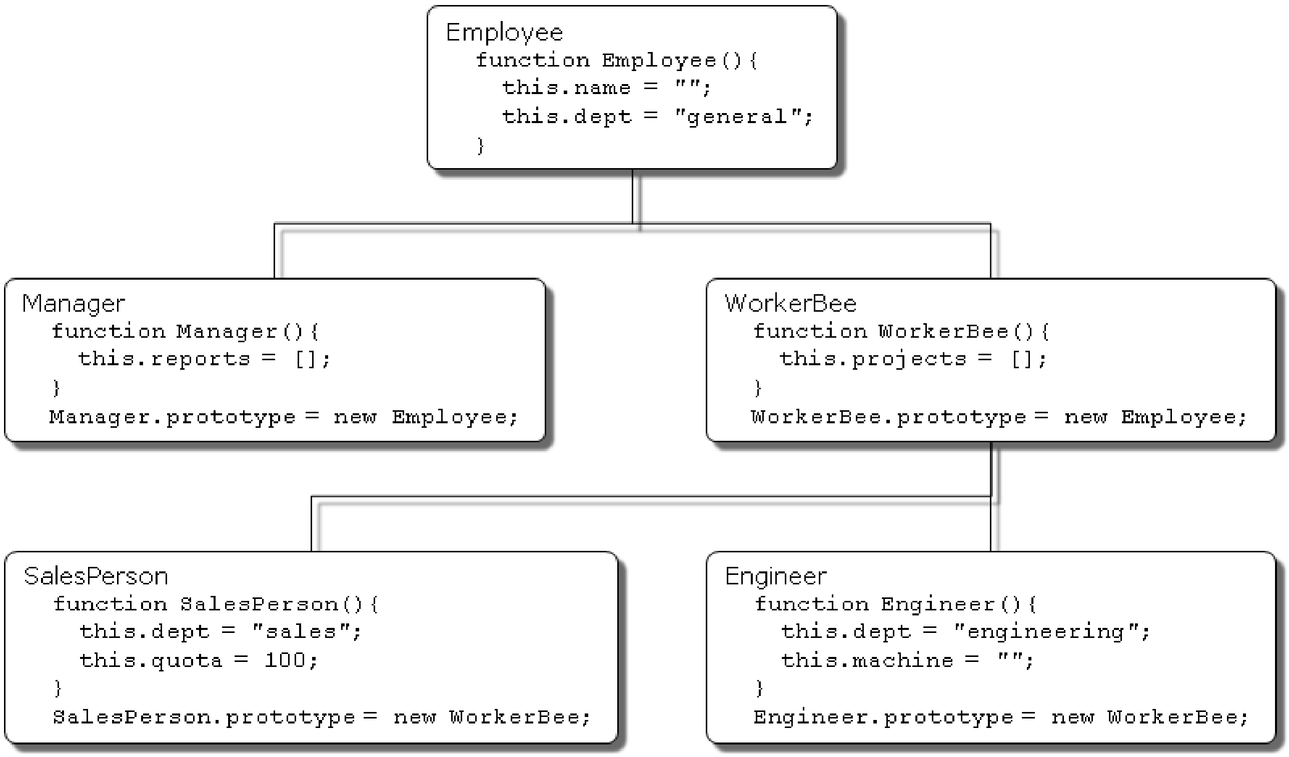
- `Employee` has the properties `name` (whose value defaults to the empty string) and `dept` (whose value defaults to "general").
- `Manager` is based on `Employee`. It adds the `reports` property (whose value defaults to an empty array, intended to have an array of `Employee` objects as its value).
- `WorkerBee` is also based on `Employee`. It adds the `projects` property (whose value defaults to an empty array, intended to have an array of strings as its value).
- `SalesPerson` is based on `WorkerBee`. It adds the `quota` property (whose value defaults to 100). It also overrides the `dept` property with the value "sales", indicating that all salespersons are in the same department.
- `Engineer` is based on `WorkerBee`. It adds the `machine` property (whose value defaults to the empty string) and also overrides the `dept` property with the value "engineering".

Creating the hierarchy

There are several ways to define appropriate constructor functions to implement the Employee hierarchy. How you choose to define them depends largely on what you want to be able to do in your application.

This section shows how to use very simple (and comparatively inflexible) definitions to demonstrate how to get the inheritance to work. In these definitions, you cannot specify any property values when you create an object. The newly-created object simply gets the default values, which you can change at a later time. Figure 8.2 illustrates the hierarchy with these simple definitions.

In a real application, you would probably define constructors that allow you to provide property values at object creation time (see More flexible constructors for information). For now, these simple definitions demonstrate how the inheritance occurs.



```
Employee
  function Employee(){
    this.name = "";
    this.dept = "general";
  }
```

```
Manager
  function Manager(){
    this.reports = [];
  }
  Manager.prototype = new Employee;
```

```
WorkerBee
  function WorkerBee(){
    this.projects = [];
  }
  WorkerBee.prototype = new Employee;
```

```
SalesPerson
  function SalesPerson(){
    this.dept = "sales";
    this.quota = 100;
  }
  SalesPerson.prototype = new WorkerBee;
```

```
Engineer
  function Engineer(){
    this.dept = "engineering";
    this.machine = "";
  }
  Engineer.prototype = new WorkerBee;
```

Figure 8.2: The Employee object definitions

Note: *Directly assigning to FunctionName.prototype removes its original prototype's "constructor" property. As a result, (new WorkerBee).constructor yields "Employee" (instead of expected "WorkerBee"). Care must be taken to preserve the original prototype's constructor. For instance, assign the parent to FunctionName.prototype.__proto__ instead. For example, WorkerBee.prototype.__proto__ = new Employee; This way, (new WorkerBee).constructor yields expected "WorkerBee".*

The following Java and JavaScript `Employee` definitions are similar. The only differences are that you need to specify the type for each property in Java but not in JavaScript, and you need to create an explicit constructor method for the Java class.

JavaScript

```
function Employee () {
  this.name = "";
  this.dept = "general";
}
```

Java

```
public class Employee {
   public String name;
   public String dept;
   public Employee () {
      this.name = "";
      this.dept = "general";
   }
}
```

The `Manager` and `WorkerBee` definitions show the difference in how to specify the next object higher in the inheritance chain. In JavaScript, you add a prototypical instance as the value of the `prototype` property of the constructor function. You can do so at any time after you define the constructor. In Java, you specify the superclass within the class definition.

You cannot change the superclass outside the class definition.

JavaScript

```javascript
function Manager () {
  this.reports = [];
}
Manager.prototype = new Employee;

function WorkerBee () {
  this.projects = [];
}
WorkerBee.prototype = new Employee;
```

Java

```java
public class Manager extends Employee {
   public Employee[] reports;
   public Manager () {
      this.reports = new Employee[0];
   }
}

public class WorkerBee extends Employee {
   public String[] projects;
   public WorkerBee () {
      this.projects = new String[0];
   }
}
```

The `Engineer` and `SalesPerson` definitions create objects that descend from `WorkerBee` and hence from `Employee`. An object of these types has properties of all the objects above it in the chain. In addition, these definitions override the inherited value of the `dept` property with new values specific to these objects.

JavaScript

```javascript
function SalesPerson () {
   this.dept = "sales";
   this.quota = 100;
}
SalesPerson.prototype = new WorkerBee;

function Engineer () {
   this.dept = "engineering";
   this.machine = "";
}
Engineer.prototype = new WorkerBee;
```

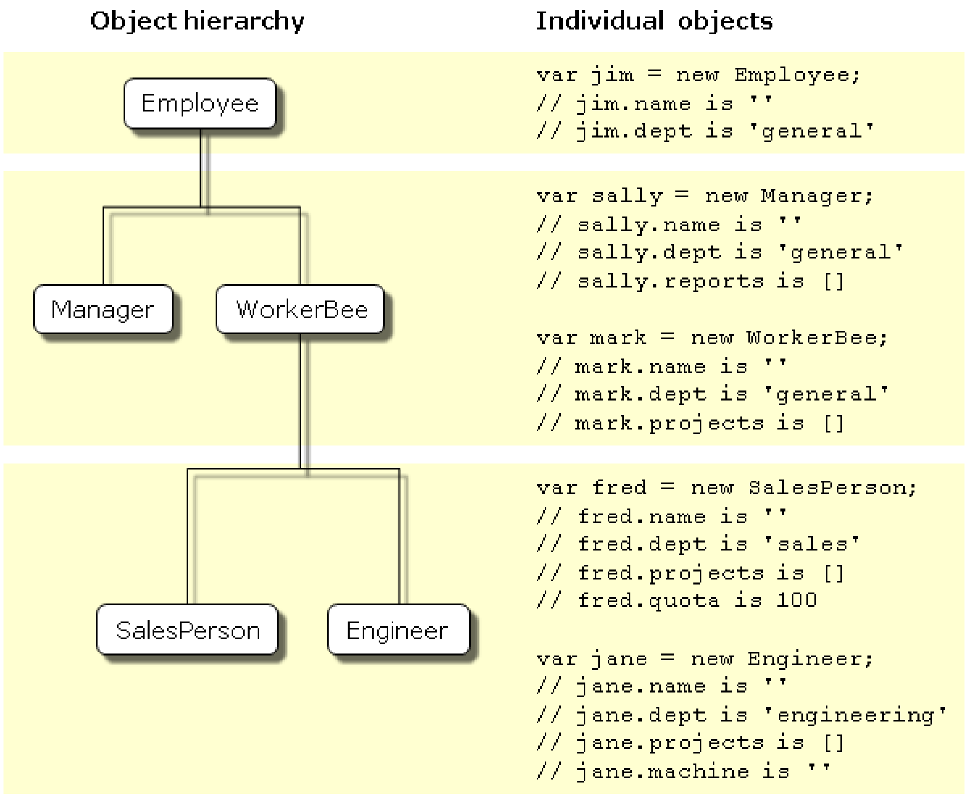
Java

```java
public class SalesPerson extends WorkerBee {
   public double quota;
   public SalesPerson () {
      this.dept = "sales";
      this.quota = 100.0;
   }
}

public class Engineer extends WorkerBee {
   public String machine;
   public Engineer () {
      this.dept = "engineering";
      this.machine = "";
   }
}
```

Using these definitions, you can create instances of these objects that get the default values for their properties. Figure 8.3 illustrates using these JavaScript definitions to create new objects and shows the property values for the new objects.

Note: *The term instance has a specific technical meaning in class-based languages. In these languages, an instance is an individual instantiation of a class and is fundamentally different from a class. In JavaScript, "instance" does not have this technical meaning because JavaScript does not have this difference between classes and instances. However, in talking about JavaScript, "instance" can be used informally to mean an object created using a particular constructor function. So, in this* *example, you could informally say that `jane` is an instance of `Engineer`. Similarly, although the terms parent, child, ancestor, and descendant do not have formal meanings in JavaScript; you can use them informally to refer to objects higher or lower in the prototype chain.*



```
var jim = new Employee;
// jim.name is ''
// jim.dept is 'general'

var sally = new Manager;
// sally.name is ''
// sally.dept is 'general'
// sally.reports is []

var mark = new WorkerBee;
// mark.name is ''
// mark.dept is 'general'
// mark.projects is []

var fred = new SalesPerson;
// fred.name is ''
// fred.dept is 'sales'
// fred.projects is []
// fred.quota is 100

var jane = new Engineer;
// jane.name is ''
// jane.dept is 'engineering'
// jane.projects is []
// jane.machine is ''
```

Figure 8.3: Creating objects with simple definitions

Object properties

This section discusses how objects inherit properties from other objects in the prototype chain and what happens when you add a property at run time.

Inheriting properties

Suppose you create the `mark` object as a `WorkerBee` (as shown in Figure 8.3) with the following statement:

```
var mark = new WorkerBee;
```

When JavaScript sees the `new` operator, it creates a new generic object and passes this new object as the value of the `this` keyword to the `WorkerBee` constructor function. The constructor function explicitly sets the value of the `projects` property, and implicitly sets the value of the internal `__proto__` property to the value of `WorkerBee.prototype`. (That property name has two underscore characters at the front and two at the end.) The `__proto__` property determines the prototype chain used to return property values. Once these properties are set, JavaScript returns the new object and the assignment statement sets the variable `mark` to that object.

This process does not explicitly put values in the `mark` object (*local* values) for the properties that `mark` inherits from the prototype chain. When you ask for the value of a property, JavaScript first checks to see if the value exists in that object. If it does, that value is returned. If the value is not there locally, JavaScript checks the prototype chain (using the `__proto__` property). If an object in the prototype chain has a value for the property, that value is returned. If no such property is found, JavaScript says the object does not have the property. In this way, the `mark` object has the following properties and values:

```
mark.name = "";
mark.dept = "general";
mark.projects = [];
```

The `mark` object inherits values for the `name` and `dept` properties from the prototypical object in `mark.__proto__`. It is assigned a local value for the `projects` property by the `WorkerBee` constructor. This gives you inheritance of properties and their values in JavaScript. Some subtleties of this process are discussed in Property inheritance revisited.

Because these constructors do not let you supply instance-specific values, this information is generic. The property values are the default ones shared by all new objects created from `WorkerBee`. You can, of course, change the values of any of these properties. So, you could give specific information for `mark` as follows:

```
mark.name = "Doe, Mark";
mark.dept = "admin";
mark.projects = ["navigator"];
```

Adding properties

In JavaScript, you can add properties to any object at run time. You are not constrained to use only the properties provided by the constructor function. To add a property that is specific to a single object, you assign a value to the object, as follows:

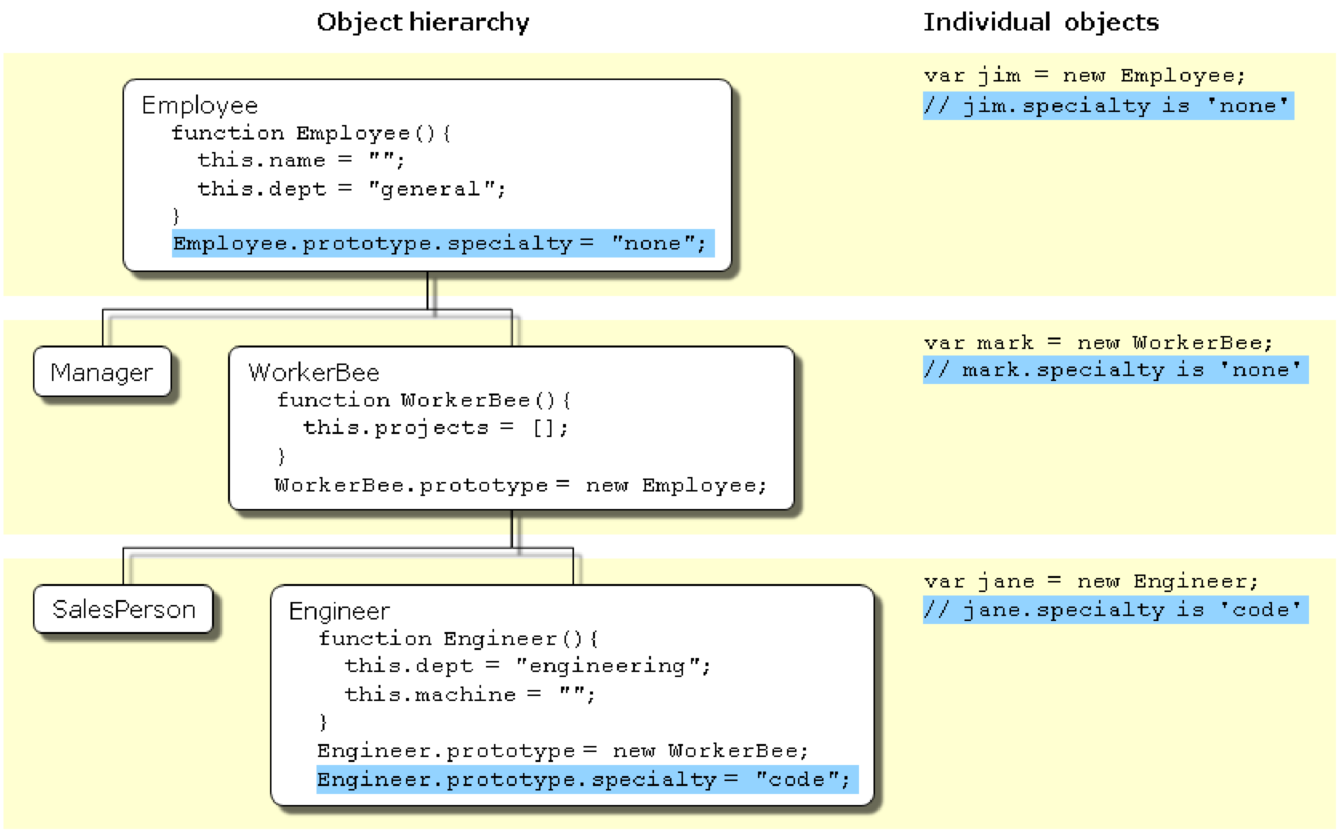
```
mark.bonus = 3000;
```

Now, the `mark` object has a `bonus` property, but no other `WorkerBee` has this property.

If you add a new property to an object that is being used as the prototype for a constructor function, you add that property to all objects that inherit properties from the prototype. For example, you can add a `specialty` property to all employees with the following statement:

```
Employee.prototype.specialty = "none";
```

As soon as JavaScript executes this statement, the `mark` object also has the `specialty` property with the value of `"none"`. The following figure shows the effect of adding this property to the `Employee` prototype and then overriding it for the `Engineer` prototype.



```
function Employee() {
  this.name = "";
  this.dept = "general";
}
Employee.prototype.specialty = "none";
```

```
function WorkerBee() {
  this.projects = [];
}
WorkerBee.prototype = new Employee;
```

```
function Engineer() {
  this.dept = "engineering";
  this.machine = "";
}
Engineer.prototype = new WorkerBee;
Engineer.prototype.specialty = "code";
```

```
var jim = new Employee;
// jim.specialty is 'none'
```

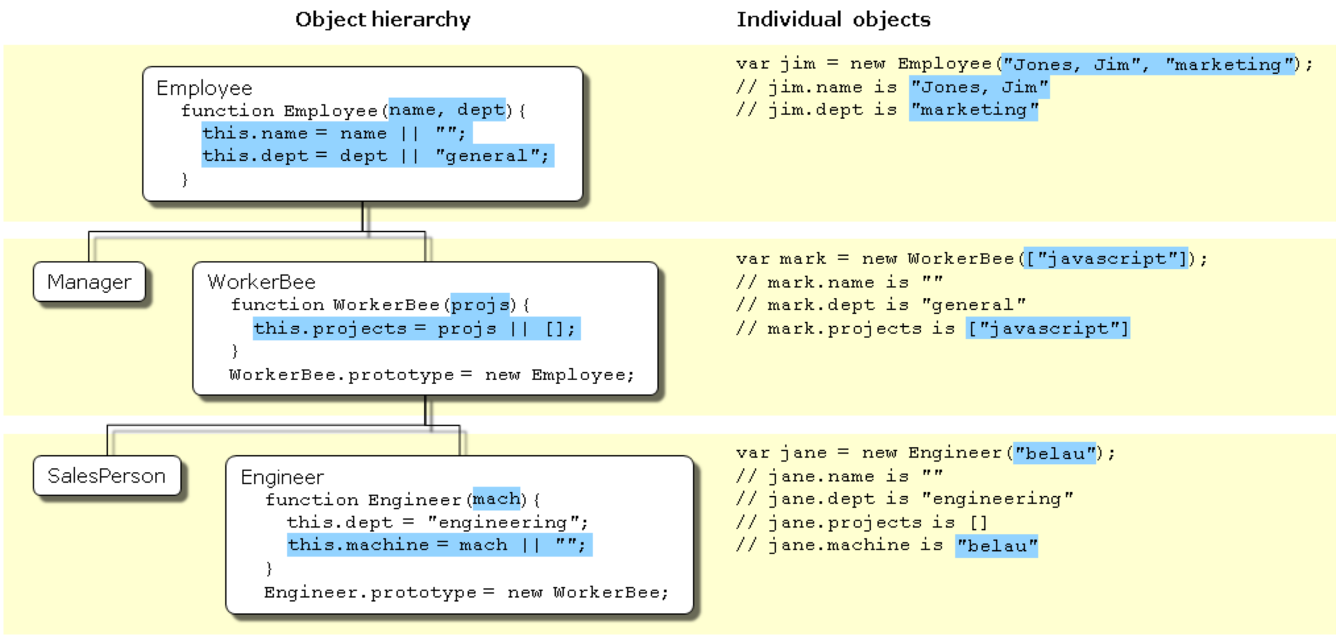
```
var mark = new WorkerBee;
// mark.specialty is 'none'
```

```
var jane = new Engineer;
// jane.specialty is 'code'
```

Figure 8.4: Adding properties

More flexible constructors

The constructor functions shown so far do not let you specify property values when you create an instance. As with Java, you can provide arguments to constructors to initialize property values for instances. The following figure shows one way to do this.



```
function Employee(name, dept) {
  this.name = name || "";
  this.dept = dept || "general";
}
```

```
function WorkerBee(projs) {
  this.projects = projs || [];
}
WorkerBee.prototype = new Employee;
```

```
function Engineer(mach) {
  this.dept = "engineering";
  this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;
```

```
var jim = new Employee("Jones, Jim", "marketing");
// jim.name is "Jones, Jim"
// jim.dept is "marketing"
```

```
var mark = new WorkerBee(["javascript"]);
// mark.name is ""
// mark.dept is "general"
// mark.projects is ["javascript"]
```

```
var jane = new Engineer("belau");
// jane.name is ""
// jane.dept is "engineering"
// jane.projects is []
// jane.machine is "belau"
```

Figure 8.5: Specifying properties in a constructor, take 1

The following table shows the Java and JavaScript definitions for these objects.

JavaScript | Java

```javascript
function Employee (name, dept) {
  this.name = name || "";
  this.dept = dept || "general";
}
```

```java
public class Employee {
   public String name;
   public String dept;
   public Employee () {
      this("", "general");
   }
   public Employee (String name) {
      this(name, "general");
   }
   public Employee (String name, String dept) {
      this.name = name;
      this.dept = dept;
   }
}
```

```javascript
function WorkerBee (projs) {
  this.projects = projs || [];
}
WorkerBee.prototype = new Employee;
```

```java
public class WorkerBee extends Employee {
   public String[] projects;
   public WorkerBee () {
      this(new String[0]);
   }
   public WorkerBee (String[] projs) {
      projects = projs;
   }
}
```

```javascript
function Engineer (mach) {
   this.dept = "engineering";
   this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;
```

```java
public class Engineer extends WorkerBee {
   public String machine;
   public Engineer () {
      dept = "engineering";
      machine = "";
   }
   public Engineer (String mach) {
      dept = "engineering";
      machine = mach;
   }
}
```

These JavaScript definitions use a special idiom for setting default values:

```javascript
this.name = name || "";
```

The JavaScript logical OR operator (`||`) evaluates its first argument. If that argument converts to true, the operator returns it. Otherwise, the operator returns the value of the second argument. Therefore, this line of code tests to see if `name` has a useful value for the `name` property. If it does, it sets `this.name` to that value. Otherwise, it sets `this.name` to the empty string. This chapter uses this idiom for brevity; however, it can be puzzling at first glance.

Note: *This may not work as expected if the constructor function is called with arguments which convert to `false` (like `0` (zero) and empty string (`""`). In this case the default value will be chosen.*

With these definitions, when you create an instance of an object, you can specify values for the locally defined properties. As shown in Figure 8.5, you can use the following statement to create a new `Engineer`:

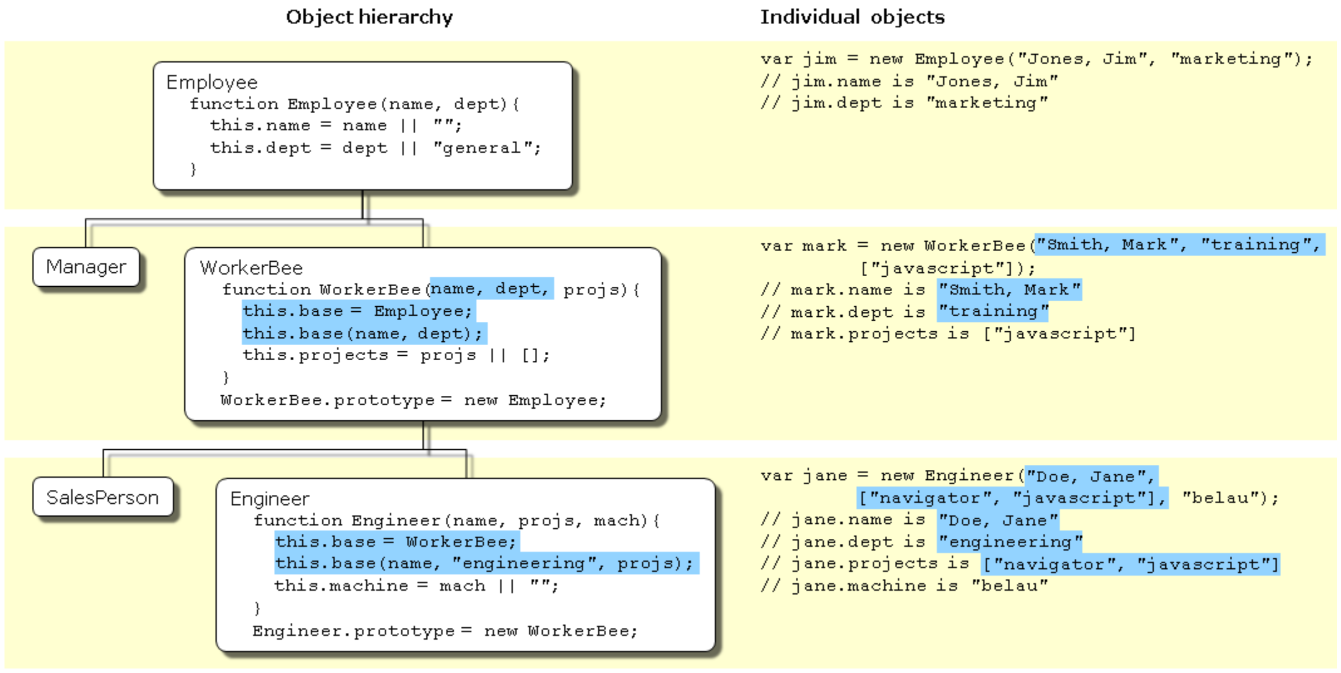
```
var jane = new Engineer("belau");
```

`Jane`'s properties are now:

```
jane.name == "";
jane.dept == "engineering";
jane.projects == [];
jane.machine == "belau"
```

Notice that with these definitions, you cannot specify an initial value for an inherited property such as `name`. If you want to specify an initial value for inherited properties in JavaScript, you need to add more code to the constructor function.

So far, the constructor function has created a generic object and then specified local properties and values for the new object. You can have the constructor add more properties by directly calling the constructor function for an object higher in the prototype chain. The following figure shows these new definitions.



```
function Employee(name, dept){
  this.name = name || "";
  this.dept = dept || "general";
}
```

```
function WorkerBee(name, dept, projs){
  this.base = Employee;
  this.base(name, dept);
  this.projects = projs || [];
}
WorkerBee.prototype = new Employee;
```

```
function Engineer(name, projs, mach){
  this.base = WorkerBee;
  this.base(name, "engineering", projs);
  this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;
```

```
var jim = new Employee("Jones, Jim", "marketing");
// jim.name is "Jones, Jim"
// jim.dept is "marketing"
```

```
var mark = new WorkerBee("Smith, Mark", "training",
           ["javascript"]);
// mark.name is "Smith, Mark"
// mark.dept is "training"
// mark.projects is ["javascript"]
```

```
var jane = new Engineer("Doe, Jane",
           ["navigator", "javascript"], "belau");
// jane.name is "Doe, Jane"
// jane.dept is "engineering"
// jane.projects is ["navigator", "javascript"]
// jane.machine is "belau"
```

Figure 8.6 Specifying properties in a constructor, take 2

Let's look at one of these definitions in detail. Here's the new definition for the `Engineer` constructor:

```
function Engineer (name, projs, mach) {
  this.base = WorkerBee;
  this.base(name, "engineering", projs);
  this.machine = mach || "";
}
```

Suppose you create a new `Engineer` object as follows:

```
var jane = new Engineer("Doe, Jane", ["navigator", "javascript"], "belau");
```

JavaScript follows these steps:

1. The `new` operator creates a generic object and sets its `__proto__` property to `Engineer.prototype`.
2. The `new` operator passes the new object to the `Engineer` constructor as the value of the `this` keyword.
3. The constructor creates a new property called `base` for that object and assigns the value of the `WorkerBee` constructor to the `base` property. This makes the `WorkerBee` constructor a method of the `Engineer` object.The name of the `base` property is not special. You can use any legal property name; `base` is simply evocative of its purpose.
4. The constructor calls the `base` method, passing as its arguments two of the arguments passed to the constructor (`"Doe, Jane"` and `["navigator", "javascript"]`) and also the string `"engineering"`. Explicitly using `"engineering"` in the constructor indicates that all `Engineer` objects have the same value for the inherited `dept` property, and this value overrides the value inherited from `Employee`.

5. Because `base` is a method of `Engineer`, within the call to `base`, JavaScript binds the `this` keyword to the object created in Step 1. Thus, the `WorkerBee` function in turn passes the "`Doe, Jane`" and "`engineering`" arguments to the `Employee` constructor function. Upon return from the `Employee` constructor function, the `WorkerBee` function uses the remaining argument to set the `projects` property.
6. Upon return from the `base` method, the `Engineer` constructor initializes the object's `machine` property to "`belau`".
7. Upon return from the constructor, JavaScript assigns the new object to the `jane` variable.

You might think that, having called the `WorkerBee` constructor from inside the `Engineer` constructor, you have set up inheritance appropriately for `Engineer` objects. This is not the case. Calling the `WorkerBee` constructor ensures that an `Engineer` object starts out with the properties specified in all constructor functions that are called. However, if you later add properties to the `Employee` or `WorkerBee` prototypes, those properties are not inherited by the `Engineer` object. For example, assume you have the following statements:

```
function Engineer (name, projs, mach) {
  this.base = WorkerBee;
  this.base(name, "engineering", projs);
  this.machine = mach || "";
}
var jane = new Engineer("Doe, Jane", ["navigator", "javascript"], "belau");
Employee.prototype.specialty = "none";
```

The `jane` object does not inherit the `specialty` property. You still need to explicitly set up the prototype to ensure dynamic inheritance. Assume instead you have these statements:

```
function Engineer (name, projs, mach) {
  this.base = WorkerBee;
  this.base(name, "engineering", projs);
  this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;
var jane = new Engineer("Doe, Jane", ["navigator", "javascript"], "belau");
Employee.prototype.specialty = "none";
```

Now the value of the `jane` object's `specialty` property is "none".

Another way of inheriting is by using the `call()` / `apply()` methods. Below are equivalent:

```
function Engineer (name, projs, mach) {
  this.base = WorkerBee;
  this.base(name, "engineering", projs);
  this.machine = mach || "";
}
```

```
function Engineer (name, projs, mach) {
  WorkerBee.call(this, name, "engineering", projs);
  this.machine = mach || "";
}
```

Using the javascript `call()` method makes a cleaner implementation because the `base` is not needed anymore.

Property inheritance revisited

The preceding sections described how JavaScript constructors and prototypes provide hierarchies and inheritance. This section discusses some subtleties that were not necessarily apparent in the earlier discussions.

Local versus inherited values

When you access an object property, JavaScript performs these steps, as described earlier in this chapter:

1. Check to see if the value exists locally. If it does, return that value.
2. If there is not a local value, check the prototype chain (using the `__proto__` property).
3. If an object in the prototype chain has a value for the specified property, return that value.
4. If no such property is found, the object does not have the property.

The outcome of these steps depends on how you define things along the way. The original example had these definitions:

```
function Employee () {
  this.name = "";
  this.dept = "general";
}

function WorkerBee () {
  this.projects = [];
}
WorkerBee.prototype = new Employee;
```

With these definitions, suppose you create `amy` as an instance of `WorkerBee` with the following statement:

```
var amy = new WorkerBee;
```

The `amy` object has one local property, `projects`. The values for the `name` and `dept` properties are not local to `amy` and so are gotten from the `amy` object's `__proto__` property. Thus, `amy` has these property values:

```
amy.name == "";
amy.dept == "general";
amy.projects == [];
```

Now suppose you change the value of the `name` property in the prototype associated with `Employee`:

```
Employee.prototype.name = "Unknown"
```

At first glance, you might expect that new value to propagate down to all the instances of `Employee`. However, it does not.

When you create *any* instance of the `Employee` object, that instance gets a local value for the `name` property (the empty string). This means that when you set the `WorkerBee` prototype by creating a new `Employee` object, `WorkerBee.prototype` has a local value for the `name` property. Therefore, when JavaScript looks up the `name` property of the `amy` object (an instance of `WorkerBee`), JavaScript finds the local value for that property in `WorkerBee.prototype`. It therefore does not look farther up the chain to `Employee.prototype`.

If you want to change the value of an object property at run time and have the new value be inherited by all descendants of the object, you cannot define the property in the object's constructor function. Instead, you add it to the constructor's associated prototype. For example, assume you change the preceding code to the following:

```
function Employee () {
  this.dept = "general";
}
Employee.prototype.name = "";

function WorkerBee () {
  this.projects = [];
}
WorkerBee.prototype = new Employee;

var amy = new WorkerBee;

Employee.prototype.name = "Unknown";
```

In this case, the `name` property of `amy` becomes "Unknown".

As these examples show, if you want to have default values for object properties and you want to be able to change the default values at run time, you should set the properties in the constructor's prototype, not in the constructor function itself.

Determining instance relationships

Property lookup in JavaScript looks within an object's own properties and, if the property name is not found, it looks within the special object property `__proto__`. This continues recursively; the process is called "lookup in the prototype chain".

The special property `__proto__` is set when an object is constructed; it is set to the value of the constructor's `prototype` property. So the expression `new Foo()` creates an object with `__proto__ == Foo.prototype`. Consequently, changes to the properties of `Foo.prototype` alters the property lookup for all objects that were created by `new Foo()`.

Every object has a `__proto__` object property (except `Object`); every function has a `prototype` object property. So objects can be related by 'prototype inheritance' to other objects. You can test for inheritance by comparing an object's `__proto__` to a function's `prototype` object. JavaScript provides a shortcut: the `instanceof` operator tests an object against a function and returns true if the object inherits from the function prototype. For example,

```
var f = new Foo();
var isTrue = (f instanceof Foo);
```

For a more detailed example, suppose you have the same set of definitions shown in Inheriting properties. Create an `Engineer` object as follows:

```
var chris = new Engineer("Pigman, Chris", ["jsd"], "fiji");
```

With this object, the following statements are all true:

```
chris.__proto__ == Engineer.prototype;
chris.__proto__.__proto__ == WorkerBee.prototype;
chris.__proto__.__proto__.__proto__ == Employee.prototype;
chris.__proto__.__proto__.__proto__.__proto__ == Object.prototype;
chris.__proto__.__proto__.__proto__.__proto__.__proto__ == null;
```

Given this, you could write an `instanceOf` function as follows:

```
function instanceOf(object, constructor) {
   while (object != null) {
      if (object == constructor.prototype)
         return true;
      if (typeof object == 'xml') {
        return constructor.prototype == XML.prototype;
      }
      object = object.__proto__;
   }
   return false;
}
```

Note: *The implementation above checks the type of the object against "xml" in order to work around a quirk of how XML objects are represented in recent versions of JavaScript. See bug 634150 if you want the nitty-gritty details.*

Using the `instanceOf` function defined above, these expressions are true:

```
instanceOf (chris, Engineer)
instanceOf (chris, WorkerBee)
instanceOf (chris, Employee)
instanceOf (chris, Object)
```

But the following expression is false:

```
instanceOf (chris, SalesPerson)
```

Global information in constructors

When you create constructors, you need to be careful if you set global information in the constructor. For example, assume that you want a unique ID to be automatically assigned to each new employee. You could use the following definition for `Employee`:

```
var idCounter = 1;

function Employee (name, dept) {
   this.name = name || "";
   this.dept = dept || "general";
   this.id = idCounter++;
}
```

With this definition, when you create a new `Employee`, the constructor assigns it the next ID in sequence and then increments the global ID counter. So, if your next statement is the following, `victoria.id` is 1 and `harry.id` is 2:

```
var victoria = new Employee("Pigbert, Victoria", "pubs")
var harry = new Employee("Tschopik, Harry", "sales")
```

At first glance that seems fine. However, `idCounter` gets incremented every time an `Employee` object is created, for whatever purpose. If you create the entire `Employee` hierarchy shown in this chapter, the `Employee` constructor is called every time you set up a prototype. Suppose you have the following code:

```
var idCounter = 1;

function Employee (name, dept) {
   this.name = name || "";
   this.dept = dept || "general";
   this.id = idCounter++;
}

function Manager (name, dept, reports) {...}
Manager.prototype = new Employee;

function WorkerBee (name, dept, projs) {...}
WorkerBee.prototype = new Employee;

function Engineer (name, projs, mach) {...}
Engineer.prototype = new WorkerBee;

function SalesPerson (name, projs, quota) {...}
SalesPerson.prototype = new WorkerBee;

var mac = new Engineer("Wood, Mac");
```

Further assume that the definitions omitted here have the `base` property and call the constructor above them in the prototype chain. In this case, by the time the `mac` object is created, `mac.id` is 5.

Depending on the application, it may or may not matter that the counter has been incremented these extra times. If you care about the exact value of this counter, one possible solution involves instead using the following constructor:

```
function Employee (name, dept) {
   this.name = name || "";
   this.dept = dept || "general";
   if (name)
      this.id = idCounter++;
}
```

When you create an instance of `Employee` to use as a prototype, you do not supply arguments to the constructor. Using this definition of the constructor, when you do not supply arguments, the constructor does not assign a value to the id and does not update the counter. Therefore, for an `Employee` to get an assigned id, you must specify a name for the employee. In this example, `mac.id` would be 1.

No multiple inheritance

Some object-oriented languages allow multiple inheritance. That is, an object can inherit the properties and values from unrelated parent objects. JavaScript does not support multiple inheritance.

Inheritance of property values occurs at run time by JavaScript searching the prototype chain of an object to find a value. Because an object has a single associated prototype, JavaScript cannot dynamically inherit from more than one prototype chain.

In JavaScript, you can have a constructor function call more than one other constructor function within it. This gives the illusion of multiple inheritance. For example, consider the following statements:

```
function Hobbyist (hobby) {
   this.hobby = hobby || "scuba";
}

function Engineer (name, projs, mach, hobby) {
   this.base1 = WorkerBee;
   this.base1(name, "engineering", projs);
   this.base2 = Hobbyist;
   this.base2(hobby);
   this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;

var dennis = new Engineer("Doe, Dennis", ["collabra"], "hugo")
```

Further assume that the definition of `WorkerBee` is as used earlier in this chapter. In this case, the `dennis` object has these properties:

```
dennis.name == "Doe, Dennis"
dennis.dept == "engineering"
dennis.projects == ["collabra"]
dennis.machine == "hugo"
dennis.hobby == "scuba"
```

So `dennis` does get the `hobby` property from the `Hobbyist` constructor. However, assume you then add a property to the `Hobbyist` constructor's prototype:

```
Hobbyist.prototype.equipment = ["mask", "fins", "regulator", "bcd"]
```

The `dennis` object does not inherit this new property.

« Previous Next »

Help us improve the quality of our code samples by answering this brief survey: http://bit.ly/sample-demo

Inheritance revisited

This article is in need of a technical review.

See Inheritance and the constructor's prototype for a description of JavaScript inheritance and the constructor's prototype.

Inheritance has always been available in JavaScript, but the examples on this page use some methods introduced in ECMAScript 5. See the different method pages to see if they can be emulated.

Example

`B` shall inherit from `A`:

```
function A(a){
  this.varA = a;
}

// What is the purpose of including varA in the prototype when A.prototype.varA w
  varA : null,  // Shouldn't we strike varA from the prototype as doing nothing?
    // ...  }
}

function B(a, b){
  A.call(this, a);
  this.varB = b;
}
B.prototype = Object.create(A.prototype, {
  varB : {
    value: null,
    enumerable: true,
    configurable: true,
    writable: true
  },
  doSomething : {
    value: function(){ // override      A.prototype.doSomething.apply(this, argum
    enumerable: true,
    configurable: true,
    writable: true
  }
});

var b = new B();
b.doSomething();

```

The important parts are:

- Types are defined in `.prototype`
- You use `Object.create()` to inherit

`prototype` and Object.getPrototypeOf

JavaScript is a bit confusing for developers coming from Java or C++, as it's all dynamic, all runtime, and it has no classes at all. It's all just instances (objects). Even the "classes" we simulate are just a function object.

You probably already noticed that our `function A` has a special property called `prototype`. This special property works with the JavaScript `new` operator. The reference to the prototype object is copied to the internal `[[Prototype]]` property of the new instance. For example, when you do `var a1 = new A()`, JavaScript (after creating the object in memory and before running function `A()` with `this` defined to it) sets `a1.[[Prototype]] = A.prototype`. When you then access properties of the instance, JavaScript first checks whether they exist on that object directly, and if not, it looks in `[[Prototype]]`. This means that all the stuff you define in `prototype` is effectively shared by all instances, and you can even later change parts of `prototype` and have the changes appear in all existing instances, if you wanted to.

If, in the example above, you do `var a1 = new A(); var a2 = new A();` then `a1.doSomething` would actually refer to `Object.getPrototypeOf(a1).doSomething`, which is the same as the `A.prototype.doSomething` you defined, i.e. `Object.getPrototypeOf(a1).doSomething == Object.getPrototypeOf(a2).doSomething == A.prototype.doSomething`.

In short, `prototype` is for types, while `Object.getPrototypeOf()` is the same for instances.

`[[Prototype]]` is looked at *recursively*, i.e. `a1.doSomething`, `Object.getPrototypeOf(a1).doSomething`, `Object.getPrototypeOf(Object.getPrototypeOf(a1)).doSomething` etc., until it's found or `Object.getPrototypeOf` returns null.

So, when you call

```
var o = new Foo();
```

JavaScript actually just does

```
var o = new Object();
o.[[Prototype]] = Foo.prototype;
o.Foo();
```

(or something like that) and when you later do

```
o.someProp;
```

it checks whether `o` has a property `someProp`. If not it checks `Object.getPrototypeOf(o).someProp` and if that doesn't exist it checks `Object.getPrototypeOf(Object.getPrototypeOf(o)).someProp` and so on.

« Previous Next »

Help us improve the quality of our code samples by answering this brief survey: http://bit.ly/sample-demo

Iterators and generators

by 18 contributors: Show all...

The content of this page is outdated and Firefox-specific. Other pages document the ECMAScript 6 iterator protocol as well as generators.

Introduced in JavaScript 1.7

Processing each of the items in a collection is a very common operation. JavaScript provides a number of ways of iterating over a collection, from simple `for` and `for each` loops to `map()`, `filter()` and array comprehensions. Iterators and Generators, introduced in JavaScript 1.7, bring the concept of iteration directly into the core language and provide a mechanism for customizing the behavior of `for...in` and `for each` loops.

Iterators

An Iterator is an object that knows how to access items from a collection one at a time, while keeping track of its current position within that sequence. In JavaScript an iterator is an object that provides a `next()` method which returns the next item in the sequence. This method can optionally raise a `StopIteration` exception when the sequence is exhausted.

Once created, an iterator object can be used either explicitly by repeatedly calling `next()`, or implicitly using JavaScript's `for...in` and `for each` constructs.

Simple iterators for objects and arrays can be created using the `Iterator()` function:

```
var lang = { name: 'JavaScript', birthYear: 1995 };
var it = Iterator(lang);
```

Once initialized, the `next()` method can be called to access key-value pairs from the object in turn:

```
var pair = it.next(); // Pair is ["name", "JavaScript"]pair = it.next(); // Pair
 // A StopIteration exception is thrown

```

A `for...in` loop can be used instead of calling the `next()` method directly. The loop will automatically terminate when the `StopIteration` exception is raised.

```
var it = Iterator(lang);
for (var pair in it)
  print(pair); // prints each [key, value] pair in turn
```

If we just want to iterate over the object's keys, we can pass a second argument of `true` to the `Iterator()` function:

```
var it = Iterator(lang, true);
for (var key in it)
  print(key); // prints each key in turn
```

One advantage of using `Iterator()` to access the contents of an object is that custom properties that have been added to `Object.prototype` will not be included in the sequence.

`Iterator()` can be used with arrays as well:

```
var langs = ['JavaScript', 'Python', 'C++'];
var it = Iterator(langs);
for (var pair in it)
  print(pair); // prints each [index, language] pair in turn
```

As with objects, passing `true` as the second argument will result in iteration occurring over the array indices:

```
var langs = ['JavaScript', 'Python', 'C++'];
var it = Iterator(langs, true);
for (var i in it)
  print(i); // prints 0, then 1, then 2
```

It is also possible to assign block scoped variables to both index and value within the for loop using the `let` keyword and a destructuring assignment:

```
var langs = ['JavaScript', 'Python', 'C++'];
var it = Iterator(langs);
for (let [i, lang] in it)
 print(i + ': ' + lang); // prints "0: JavaScript" etc.
```

Defining custom iterators

Some objects represent collections of items that should be iterated over in a specific way.

- Iterating over a range object should return the numbers in that range one by one.
- The leaves in a tree can be visited using depth-first or breadth-first traversal.
- Iterating over an object representing the results from a database query should return rows one by one, even if the entire result set has not been loaded in to a single array.
- An iterator on an infinite mathematical sequence (such as the Fibonacci sequence) should be able to return results one by one without creating an infinite length data structure.

JavaScript lets you write code that represents custom iteration logic and link it to an object.

We'll create a simple `Range` object which stores a low and high value.

```
function Range(low, high){
  this.low = low;
  this.high = high;
}
```

Now we'll create a custom iterator that can return a sequence of inclusive integers from that range. The iterator interface requires that we provide a `next()` method which either returns an item from the sequence or throws a `StopIteration` exception.

```
function RangeIterator(range){
  this.range = range;
  this.current = this.range.low;
}
RangeIterator.prototype.next = function(){
  if (this.current > this.range.high)
    throw StopIteration;
  else
    return this.current++;
};
```

Our `RangeIterator` is instantiated with a range instance, and maintains its own `current` property to track how far along in the sequence it has got.

Finally, to associate our `RangeIterator` with the `Range` object we need to add a special `__iterator__` method to `Range`. This will be called when we attempt to iterate over a `Range` instance, and should return an instance of `RangeIterator`, which implements the iterator logic.

```
Range.prototype.__iterator__ = function(){
  return new RangeIterator(this);
};
```

Having hooked in our custom iterator, we can iterate over a range instance with the following:

```
var range = new Range(3, 5);
for (var i in range)
  print(i); // prints 3, then 4, then 5 in sequence
```

Generators: a better way to build Iterators

While custom iterators are a useful tool, their creation requires careful programming due to the need to explicitly maintain their internal state. Generators provide a powerful alternative: they allow you to define an iterative algorithm by writing a single function which can maintain its own state.

A generator is a special type of function that works as a factory for iterators. A function becomes a generator if it contains one or more `yield` expressions.

Note: *The `yield` keyword is only available to code blocks in HTML wrapped in a `<script type="application/javascript;version=1.7">` block (or higher version). XUL script tags have access to these features without needing this special block.*

When a generator function is called the body of the function does not execute straight away; instead, it returns a generator-iterator object. Each call to the generator-iterator's `next()` method will execute the body of the function up to the next `yield` expression and return its result. When either the end of the function or a `return` statement is reached, a `StopIteration` exception is thrown.

This is best illustrated with an example:

```
function simpleGenerator(){
  yield "first";
  yield "second";
  yield "third";
  for (var i = 0; i < 3; i++)
    yield i;
}

var g = simpleGenerator();
print(g.next()); // prints "first"print(g.next()); // prints "second"print(g.next
 // StopIteration is thrown

```

A generator function can be used directly as the `__iterator__` method of a class, greatly reducing the amount of code needed to create custom iterators. Here is our `Range` rewritten to use a generator:

```
function Range(low, high){
  this.low = low;
  this.high = high;
}
Range.prototype.__iterator__ = function(){
  for (var i = this.low; i <= this.high; i++)
    yield i;
};
var range = new Range(3, 5);
for (var i in range)
  print(i); // prints 3, then 4, then 5 in sequence
```

Not all generators terminate; it is possible to create a generator that represents an infinite sequence. The following generator implements the Fibonacci sequence, where each element is the sum of the two previous elements:

```
function fibonacci(){
  var fn1 = 1;
  var fn2 = 1;
  while (1){
    var current = fn2;
    fn2 = fn1;
    fn1 = fn1 + current;
    yield current;
  }
}

var sequence = fibonacci();
print(sequence.next()); // 1print(sequence.next()); // 1print(sequence.next()); /
 // 13

```

Generator functions can take arguments, which are bound the first time the function is called. Generators can be terminated (causing them to raise a `StopIteration` exception) using a `return` statement. The following `fibonacci()` variant takes an optional limit argument, and will terminate once that limit has been passed.

```
function fibonacci(limit){
  var fn1 = 1;
  var fn2 = 1;
  while (1){
    var current = fn2;
    fn2 = fn1;
    fn1 = fn1 + current;
    if (limit && current > limit)
      return;
    yield current;
  }
}
```

Advanced generators

Generators compute their yielded values on demand, which allows them to efficiently represent sequences that are expensive to compute, or even infinite sequences as demonstrated above.

In addition to the `next()` method, generator-iterator objects also have a `send()` method which can be used to modify the internal state of the generator. A value passed to `send()` will be treated as the result of the last `yield` expression that paused the generator. You must start a generator by calling `next()` at least once before you can use `send()` to pass in a specific value.

Here is the fibonacci generator using `send()` to restart the sequence:

```
function fibonacci(){
  var fn1 = 1;
  var fn2 = 1;
  while (1){
    var current = fn2;
    fn2 = fn1;
    fn1 = fn1 + current;
    var reset = yield current;
    if (reset){
        fn1 = 1;
        fn2 = 1;
    }
  }
}

var sequence = fibonacci();
print(sequence.next());     // 1print(sequence.next());     // 1print(sequence.ne
 // 3

```

> ***Note:*** *As a point of interest, calling* `send(undefined)` *is equivalent to calling* `next()`*. However, starting a newborn generator with any value other than undefined when calling* `send()` *will result in a* `TypeError` *exception.*

You can force a generator to throw an exception by calling its `throw()` method and passing the exception value it should throw. This exception will be thrown from the current suspended context of the generator, as if the `yield` that is currently suspended were instead a `throw value` statement.

If a `yield` is not encountered during the processing of the thrown exception, then the exception will propagate up through the call to `throw()`, and subsequent calls to `next()` will result in `StopIteration` being thrown.

Generators have a `close()` method that forces the generator to close itself. The effects of closing a generator are:

1. Any `finally` clauses active in the generator function are run.
2. If a `finally` clause throws any exception other than `StopIteration`, the exception is propagated to the caller of the `close()` method.
3. The generator terminates.

Generator expressions

Introduced in JavaScript 1.8

A significant drawback of array comprehensions is that they cause an entire new array to be constructed in memory. When the input to the comprehension is itself a small array the overhead involved is insignificant — but when the input is a large array or an expensive (or indeed infinite) generator the creation of a new array can be problematic.

Generators enable lazy computation of sequences, with items calculated on-demand as they are needed. *Generator expressions* are syntactically almost identical to array comprehensions — they use parentheses instead of braces (and `for...in` instead of `for each...in`) — but instead of building an array they create a generator that can execute lazily. You can think of them as short hand syntax for creating generators.

Suppose we have an iterator `it` which iterates over a large sequence of integers. We want to create a new iterator that will iterate over their doubles. An array comprehension would create a full array in memory containing the doubled values:

```
var doubles = [i * 2 for (i in it)];
```

A generator expression on the other hand would create a new iterator which would create doubled values on demand as they were needed:

```
var it2 = (i * 2 for (i in it));
print(it2.next()); // The first value from it, doubledprint(it2.next());
 // The second value from it, doubled
```

When a generator expression is used as the argument to a function, the parentheses used for the function call means that the outer parentheses can be omitted:

```
var result = doSomething(i * 2 for (i in it));
```

« Previous Next »

Help us improve the quality of our code samples by answering this brief survey: http://bit.ly/sample-demo

Closures

by 42 contributors: Show all...

Closures are functions that refer to independent (free) variables.

In other words, the function defined in the closure 'remembers' the environment in which it was created.

Consider the following:

```
function init() {
    var name = "Mozilla"; // name is a local variable created by init    functior
    displayName();
}
init();

```

`init()` creates a local variable `name` and then a function called `displayName()`. `displayName()` is the inner function (a *closure*) — it is defined inside `init()`, and only available within the body of that function . Unlike `init()`, `displayName()` has no local variables of its own, and instead reuses the variable `name` declared in the parent function.

Run the code and see that this works. This is an example of *lexical scoping*: in JavaScript, the scope of a variable is defined by its location within the source code (it is apparent *lexically*) and nested functions have access to variables declared in their outer scope.

Now consider the following example:

```
function makeFunc() {
  var name = "Mozilla";
  function displayName() {
    alert(name);
  }
  return displayName;
}

var myFunc = makeFunc();
myFunc();
```

If you run this code it will have exactly the same effect as the previous `init()` example: the string "Mozilla" will be displayed in a JavaScript alert box. What's different — and interesting — is that the `displayName()` inner function was returned from the outer function before being executed.

That the code still works may seem unintuitive. Normally, the local variables within a function only exist for the duration of that function's execution. Once `makeFunc()` has finished executing, it is reasonable to expect that the name variable will no longer be accessible. Since the code still works as expected, this is obviously not the case.

The solution to this puzzle is that `myFunc` has become a *closure*. A closure is a special kind of object that combines two things: a function, and the environment in which that function was created. The environment consists of any local variables that were in-scope at the time that the closure was created. In this case, `myFunc` is a closure that incorporates both the `displayName` function and the "Mozilla" string that existed when the closure was created.

Here's a slightly more interesting example — a `makeAdder` function:

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));  // 7console.log(add10(2)); // 12
```

In this example, we have defined a function `makeAdder(x)` which takes a single argument `x` and returns a new function. The function it returns takes a single argument `y`, and returns the sum of `x` and `y`.

In essence, `makeAdder` is a function factory — it creates functions which can add a specific value to their argument. In the above example we use our function factory to create two new functions — one that adds 5 to its argument, and one that adds 10.

`add5` and `add10` are both closures. They share the same function body definition, but store different environments. In `add5`'s environment, `x` is 5. As far as `add10` is concerned, `x` is 10.

Practical closures

That's the theory out of the way — but are closures actually useful? Let's consider their practical implications. A closure lets you associate some data (the environment) with a function that operates on that data. This has obvious parallels to object oriented programming, where objects allow us to associate some data (the object's properties) with one or more methods.

Consequently, you can use a closure anywhere that you might normally use an object with only a single method.

Situations where you might want to do this are particularly common on the web. Much of the code we write in web JavaScript is event-based — we define some behavior, then attach it to an event that is triggered by the user (such as a click or a keypress). Our code is generally attached as a callback: a single function which is executed in response to the event.

Here's a practical example: suppose we wish to add some buttons to a page that adjust the text size. One way of doing this is to specify the font-size of the body element in pixels, then set the size of the other elements on the page (such as headers) using the relative em unit:

```css
body {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 12px;
}

h1 {
  font-size: 1.5em;
}
h2 {
  font-size: 1.2em;
}
```

Our interactive text size buttons can change the font-size property of the body element, and the adjustments will be picked up by other elements on the page thanks to the relative units.

Here's the JavaScript:

```
function makeSizer(size) {
  return function() {
    document.body.style.fontSize = size + 'px';
  };
}

var size12 = makeSizer(12);
var size14 = makeSizer(14);
var size16 = makeSizer(16);
```

`size12`, `size14`, and `size16` are now functions which will resize the body text to 12, 14, and 16 pixels, respectively. We can attach them to buttons (in this case links) as follows:

```
document.getElementById('size-12').onclick = size12;
document.getElementById('size-14').onclick = size14;
document.getElementById('size-16').onclick = size16;
```

```
<a href="#" id="size-12">12</a>
<a href="#" id="size-14">14</a>
<a href="#" id="size-16">16</a>
```

VIEW ON JSFIDDLE

Emulating private methods with closures

Languages such as Java provide the ability to declare methods private, meaning that they can only be called by other methods in the same class.

JavaScript does not provide a native way of doing this, but it is possible to emulate private methods using closures. Private methods aren't just useful for restricting access to code: they also provide a powerful way of managing your global namespace, keeping non-essential methods from cluttering up the public interface to your code.

Here's how to define some public functions that can access private functions and variables, using closures which is also known as the module pattern:

```
var counter = (function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  };
})();

alert(counter.value()); /* Alerts 0 */
counter.increment();
counter.increment();
alert(counter.value()); /* Alerts 2 */
counter.decrement();
alert(counter.value()); /* Alerts 1 */
```

There's a lot going on here. In previous examples each closure has had its own environment; here we create a single environment which is shared by three functions: `counter.increment`, `counter.decrement`, and `counter.value`.

The shared environment is created in the body of an anonymous function, which is executed as soon as it has been defined. The environment contains two private items: a variable called `privateCounter` and a function called `changeBy`. Neither of these private items can be accessed directly from outside the anonymous function. Instead, they must be accessed by the three public functions that are returned from the anonymous wrapper.

Those three public functions are closures that share the same environment. Thanks to JavaScript's lexical scoping, they each have access to the `privateCounter` variable and `changeBy` function.

You'll notice we're defining an anonymous function that creates a counter, and then we call it immediately and assign the result to the `Counter` variable. We could store this function in a separate variable and use it to create several counters.

```
var makeCounter = function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  }
};

var counter1 = makeCounter();
var counter2 = makeCounter();
alert(counter1.value()); /* Alerts 0 */
counter1.increment();
counter1.increment();
alert(counter1.value()); /* Alerts 2 */
counter1.decrement();
alert(counter1.value()); /* Alerts 1 */
alert(counter2.value()); /* Alerts 0 */
```

Notice how each of the two counters maintains its independence from the other. Its environment during the call of the `makeCounter()` function is different each time. The closure variable `privateCounter` contains a different instance each time.

Using closures in this way provides a number of benefits that are normally associated with object oriented programming, in particular data hiding and encapsulation.

Creating closures in loops: A common mistake

Prior to the introduction of the `let` keyword in JavaScript 1.7, a common problem with closures occurred when they were created inside a loop. Consider the following example:

```
<p id="help">Helpful notes will appear here</p>
<p>E-mail: <input type="text" id="email" name="email"></p>
<p>Name: <input type="text" id="name" name="name"></p>

```

```
<p>Age: <input type="text" id="age" name="age"></p>
```

```
function showHelp(help) {
  document.getElementById('help').innerHTML = help;
}

function setupHelp() {
  var helpText = [
      {'id': 'email', 'help': 'Your e-mail address'},
      {'id': 'name', 'help': 'Your full name'},
      {'id': 'age', 'help': 'Your age (you must be over 16)'}
    ];

  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
    document.getElementById(item.id).onfocus = function() {
      showHelp(item.help);
    }
  }
}

setupHelp();
```

VIEW ON JSFIDDLE

The `helpText` array defines three helpful hints, each associated with the ID of an input field in the document. The loop cycles through these definitions, hooking up an onfocus event to each one that shows the associated help method.

If you try this code out, you'll see that it doesn't work as expected. No matter what field you focus on, the message about your age will be displayed.

The reason for this is that the functions assigned to onfocus are closures; they consist of the function definition and the captured environment from the `setupHelp` function's scope. Three closures have been created, but each one shares the same single environment. By the time the onfocus callbacks are executed, the loop has run its course and the item variable (shared by all three closures) has been left pointing to the last entry in the `helpText` list.

One solution in this case is to use more closures: in particular, to use a function factory as described earlier on:

```
function showHelp(help) {
  document.getElementById('help').innerHTML = help;
}

function makeHelpCallback(help) {
  return function() {
    showHelp(help);
  };
}

function setupHelp() {
  var helpText = [
      {'id': 'email', 'help': 'Your e-mail address'},
      {'id': 'name', 'help': 'Your full name'},
      {'id': 'age', 'help': 'Your age (you must be over 16)'}
    ];

  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
    document.getElementById(item.id).onfocus = makeHelpCallback(item.help);
  }
}

setupHelp();
```

VIEW ON JSFIDDLE

This works as expected. Rather than the callbacks all sharing a single environment, the `makeHelpCallback` function creates a new environment for each one in which `help` refers to the corresponding string from the `helpText` array.

Performance considerations

It is unwise to unnecessarily create functions within other functions if closures are not needed for a particular task, as it will negatively affect script performance both in terms of processing speed and memory consumption.

For instance, when creating a new object/class, methods should normally be associated to the object's prototype rather than defined into the object constructor. The reason is that whenever the constructor is called, the methods would get reassigned (that is, for every object creation).

Consider the following impractical but demonstrative case:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
  this.getName = function() {
    return this.name;
  };

  this.getMessage = function() {
    return this.message;
  };
}
```

The previous code does not take advantage of the benefits of closures and thus could instead be formulated:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype = {
  getName: function() {
    return this.name;
  },
  getMessage: function() {
    return this.message;
  }
};
```

However, redefining the prototype is not recommended, so the following example is even better because it appends to the existing prototype:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype.getName = function() {
  return this.name;
};
MyObject.prototype.getMessage = function() {
  return this.message;
};
```

In the two previous examples, the inherited prototype can be shared by all objects and the method definitions need not occur at every object creation. See Details of the Object Model for more details.

« Previous Next »

Help us improve the quality of our code samples by answering this brief survey: http://bit.ly/sample-demo

Sameness in JavaScript

by 4 contributors:

ES6 has three built-in facilities for determining whether some x and some y are "the same". They are: equality or "double equals" (==), strict equality or "triple equals" (===), and `Object.is`. (Note that `Object.is` was added in ES6. Both double equals and triple equals existed prior to ES6, and their behavior remains unchanged.)

Overview

For demonstration, here are the three sameness comparisons in use:

```
x == y
```

```
x === y
```

```
Object.is(x, y)
```

Briefly, double equals will perform a type conversion when comparing two things; triple equals will do the same comparison without type conversion (by simply always returning `false` if the types differ); and `Object.is` will behave the same way as triple equals, but with special handling for `NaN` and `-0` and `+0` so that the last two are not said to be the same, while `Object.is(NaN, NaN)` will be `true`. (Comparing `NaN` with `NaN` ordinarily—i.e., using either double equals or triple equals—evaluates to `false`, because IEEE 754 says so.)

Do note that the distinction between these all have to do with their handling of primitives; none of them compares whether the parameters are conceptually similar in structure. For any non-primitive objects x and y which have the same structure but are distinct objects themselves, all of the above forms will evaluate to `false`.

For example:

```
let x = { value: 17 };
let y = { value: 17 };
console.log(Object.is(x, y)); // false;console.log(x === y);        // falsecons
 // false

```

Abstract equality, strict equality, and same value

In ES5, the comparison performed by `==` is described in Section 11.9.3, The Abstract Equality Algorithm. The `===` comparison is 11.9.6, The Strict Equality Algorithm. (Go look at these. They're brief and readable. Hint: read the strict equality algorithm first.) ES5 also describes, in Section 9.12, The SameValue Algorithm for use internally by the JS engine. It's largely the same as the Strict Equality Algorithm, except that 11.9.6.4 and 9.12.4 differ in handling `Number`s. ES6 simply proposes to expose this algorithm through `Object.is`.

We can see that with double and triple equals, with the exception of doing a type check upfront in 11.9.6.1, the Strict Equality Algorithm is a subset of the Abstract Equality Algorithm, because 11.9.6.2–7 correspond to 11.9.3.1.a–f.

A model for understanding equality comparisons?

Prior to ES6, you might have said of double equals and triple equals that one is an "enhanced" version of the other. For example, someone might say that double equals is an extended version of triple equals, because the former does everything that the latter does, but with type conversion on its operands. E.g., `6 == "6"`. (Alternatively, someone might say that double equals is the baseline, and triple equals is an enhanced version, because it requires the two operands to be the same type, so it adds an extra constraint. Which one is the better model for understanding depends on how you choose to view things.)

However, this way of thinking about the built-in sameness operators is not a model that can be stretched to allow a place for ES6's `Object.is` on this "spectrum". `Object.is` isn't simply "looser" than double equals or "stricter" than triple equals, nor does it fit somewhere in between (i.e., being both stricter than double equals, but looser than triple equals). We can see from the sameness comparisons table below that this is due to the way that `Object.is` handles `NaN`. Notice that if `Object.is(NaN, NaN)` evaluated to `false`, we *could* say that it fits on the loose/strict spectrum as an even stricter form of triple equals, one that distinguishes between `-0` and `+0`. The `NaN` handling means this is untrue, however. Unfortunately, `Object.is` simply has to be thought of in terms of its specific characteristics, rather than its looseness or strictness with regard to the equality operators.

Sameness Comparisons

x	y	`==`	`===`	`Object.is`

`undefined`	`undefined`	`true`	`true`	`true`
`null`	`null`	`true`	`true`	`true`
`true`	`true`	`true`	`true`	`true`
`false`	`false`	`true`	`true`	`true`
`"foo"`	`"foo"`	`true`	`true`	`true`
`{ foo: "bar" }`	`x`	`true`	`true`	`true`
`0`	`0`	`true`	`true`	`true`
`+0`	`-0`	`true`	`true`	`false`
`0`	`false`	`true`	`false`	`false`
`""`	`false`	`true`	`false`	`false`
`""`	`0`	`true`	`false`	`false`
`"0"`	`0`	`true`	`false`	`false`
`"17"`	`17`	`true`	`false`	`false`
`[1,2]`	`"1,2"`	`true`	`false`	`false`
`new String("foo")`	`"foo"`	`true`	`false`	`false`
`null`	`undefined`	`true`	`false`	`false`
`null`	`false`	`false`	`false`	`false`
`undefined`	`false`	`false`	`false`	`false`
`{ foo: "bar" }`	`{ foo: "bar" }`	`false`	`false`	`false`
`new String("foo")`	`new String("foo")`	`false`	`false`	`false`
`0`	`null`	`false`	`false`	`false`
`0`	`NaN`	`false`	`false`	`false`

"foo"	NaN	false	false	false
NaN	NaN	false	false	true

When to use `Object.is` versus triple equals

Aside from the way it treats `NaN`, generally, the only time `Object.is`'s special behavior towards zeroes is likely to be of interest is in the pursuit of certain metaprogramming schemes, especially regarding property descriptors when it is desirable for your work to mirror some of the characteristics of `Object.defineProperty`. If your use case does not require this, it is suggested to avoid `Object.is` and use `===` instead. Even if your requirements involve having comparisons between two `NaN` values evaluate to `true`, generally it is easier to special-case the `NaN` checks (using the `isNaN` method available from previous versions of ECMAScript) than it is to work out how surrounding computations might affect the sign of any zeroes you encounter in your comparison.

Here's an inexhaustive list of built-in methods and operators that might cause a distinction between `-0` and `+0` to manifest itself in your code:

`- (unary negation)`

It's obvious that negating `0` produces `-0`. But the abstraction of an expression can cause `-0` to creep in when you don't realize it. For example, consider:

```
let stoppingForce = obj.mass * -obj.velocity
```

If `obj.velocity` is `0` (or computes to `0`), a `-0` is introduced at that place and propogates out into `stoppingForce`.

`Math.atan2`
`Math.ceil`
`Math.pow`
`Math.round`

It's possible for a `-0` to be introduced into an expression as a return value of these methods in some cases, even when no `-0` exists as one of the parameters. E.g., using `Math.pow` to raise `-Infinity` to the power of any negative, odd exponent evaluates to `-0`. Refer to the documentation for the individual methods.

`Math.floor`

`Math.max`

`Math.min`

`Math.sin`

`Math.sqrt`

`Math.tan`

It's possible to get a `-0` return value out of these methods in some cases where a `-0` exists as one of the parameters. E.g., `Math.min(-0, +0)` evalutes to `-0`. Refer to the documentation for the individual methods.

`~`

`<<`

`>>`

Each of these operators uses the ToInt32 algorithm internally. Since there is only one representation for 0 in the internal 32-bit integer type, `-0` will not survive a round trip after an inverse operation. E.g., both `Object.is(~~(-0), -0)` and `Object.is(-0 << 2 >> 2, -0)` evaluate to `false`.

Relying on `Object.is` when the signedness of zeroes is not taken into account can be hazardous. Of course, when the intent is to distinguish between `-0` and `+0`, it does exactly what's desired.