An event handler is a JavaScript function that we register with the

browser and the browser invokes when some specified type of event occurs.

* All numbers in JavaScript are represented as floating-point

values. JavaScript represents numbers using the 64-bit floating-point format defined

by the IEEE 754 standard,

# Infinity equality

var zero = 0; // Regular zero

var negz = -0; // Negative zero

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

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

Date type

var then = new Date(2010, 0, 1); // The 1st day of the 1st month of 2010

var later = new Date(2010, 0, 1, // Same day, at 5:10:30pm, local time

17, 10, 30);

var now = new Date(); // The current date and time

var elapsed = now - then; // Date subtraction: interval in milliseconds

later.getFullYear() // => 2010

later.getMonth() // => 0: zero-based months

later.getDate() // => 1: one-based days

later.getDay() // => 5: day of week. 0 is Sunday 5 is Friday.

later.getHours() // => 17: 5pm, local time

later.getUTCHours() // hours in UTC time; depends on timezone

3.1 Numbers | 35

**Core JavaScript**

later.toString() // => "Fri Jan 01 2010 17:10:30 GMT-0800 (PST)"

later.toUTCString() // => "Sat, 02 Jan 2010 01:10:30 GMT"

later.toLocaleDateString() // => "01/01/2010"

later.toLocaleTimeString() // => "05:10:30 PM"

later.toISOString() // => "2010-01-02T01:10:30.000Z"; ES5 only

# Precedence of a variable

Within the body of a function, a local variable takes precedence over a global variable

with the same name. If you declare a local variable or function parameter with the same

name as a global variable, you effectively hide the global variable:

var scope = "global"; // Declare a global variable

function checkscope() {

var scope = "local"; // Declare a local variable with the same name

return scope; // Return the local value, not the global one

}

checkscope() // => "local"

scope = "global"; // Declare a global variable, even without var.

function checkscope2() {

scope = "local"; // Oops! We just changed the global variable.

myscope = "local"; // This implicitly declares a new global variable.

return [scope, myscope]; // Return two values.

}

checkscope2() // => ["local", "local"]: has side effects!

scope // => "local": global variable has changed.

myscope // => "local": global namespace cluttered up.

# Function hoisting

var scope = "global";

function f() {

console.log(scope); // Prints "undefined", not "global"

var scope = "local"; // Variable initialized here, but defined everywhere

console.log(scope); // Prints "local"

}

# Delete variable

Variables created in this way are regular, configurable properties of

the global object and they can be deleted:

var truevar = 1; // A properly declared global variable, nondeletable.

fakevar = 2; // Creates a deletable property of the global object.

this.fakevar2 = 3; // This does the same thing.

delete truevar // => false: variable not deleted

delete fakevar // => true: variable deleted

delete this.fakevar2 // => true: variable deleted

# Arrays

[] // An empty array: no expressions inside brackets means no elements

[1+2,3+4] // A 2-element array. First element is 3, second is 7

\*\*\*

For example, the following array contains five elements, including three

undefined elements:

var sparseArray = [1,,,,5];

\*\*\*

Object initializer expressions are like array initializer expressions, but the square brackets

are replaced by curly brackets, and each subexpression is prefixed with a property

name and a colon:

var p = { x:2.3, y:-1.2 }; // An object with 2 properties

var q = {}; // An empty object with no properties

q.x = 2.3; q.y = -1.2; // Now q has the same properties as p

Object literals can be nested. For example:

var rectangle = { upperLeft: { x: 2, y: 2 },

lowerRight: { x: 4, y: 5 } };

# Property aceess syntaxis

JavaScript defines two syntaxes for property access:

*expression* . *identifier*

*expression* [ *expression* ]

var o = {x:1,y:{z:3}}; // An example object

var a = [o,4,[5,6]]; // An example array that contains the object

o.x // => 1: property x of expression o

o.y.z // => 3: property z of expression o.y

o["x"] // => 1: property x of object o

a[1] // => 4: element at index 1 of expression a

a[2]["1"] // => 6: element at index 1 of expression

# Objects

Object creation expressions are

like invocation expressions except that they are prefixed with the keyword new:

new Object()

new Point(2,3)

4.6 Object Creation Expressions | 61

**Core JavaScript**

If no arguments are passed to the constructor function in an object creation expression,

the empty pair of parentheses can be omitted:

new Object

# Precedent operators

Consider the following expression:

w = x + y\*z;

The multiplication operator \* has a higher precedence than the addition operator +, so

the multiplication is performed before the addition. Furthermore, the assignment operator

= has the lowest precedence, so the assignment is performed after all the operations

on the right side are completed.

The rules that

are important to know are these: multiplication and division are performed before addition

and subtraction, and assignment has very low precedence and is almost always

performed last.

# Operator +

1 + 2 // => 3: addition

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

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

1 + {} // => "1[object Object]": concatenation after object-to-string

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

4.8 Arithmetic Expressions | 67

**Core JavaScript**

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

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

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

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

# ++ operator

The ++ operator never

performs string concatenation: it always converts its operand to a number and

increments it. If x is the string “1”, ++x is the number 2, but x+1 is the string “11”.

# Equality operators

===

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

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

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

are equal.

4.9 Relational Expressions | 71

**Core JavaScript**

• If one or both values is NaN, they are not equal. The NaN value is never equal to any

other value, including itself! To check whether a value x is NaN, use x !== x. NaN is

the only value of x for which this expression will be true.

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

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

• If both values are strings and contain exactly the same 16-bit values (see the sidebar

in §3.2) in the same positions, they are equal. If the strings differ in length or

content, they are not equal. Two strings may have the same meaning and the same

visual appearance, but still be encoded using different sequences of 16-bit values.

JavaScript performs no Unicode normalization, and a pair of strings like this

are not considered equal to the === or to the == operators. See

String.localeCompare() in Part III for another way to compare strings.

• If both values refer to the same object, array, or function, they are equal. If they

refer to different objects they are not equal, even if both objects have identical

properties

.==

If the two values have the same type, test them for strict equality as described above.

If they are strictly equal, they are equal. If they are not strictly equal, they are not

equal.

• If the two values do not have the same type, the == operator may still consider them

equal. Use the following rules and type conversions to check for equality:

— If one value is null and the other is undefined, they are equal.

— If one value is a number and the other is a string, convert the string to a number

and try the comparison again, using the converted value.

— If either value is true, convert it to 1 and try the comparison again. If either value

is false, convert it to 0 and try the comparison again.

— If one value is an object and the other is a number or string, convert the object

to a primitive using the algorithm described in §3.8.3 and try the comparison

again. An object is converted to a primitive value by either its toString() method

or its valueOf() method. The built-in classes of core JavaScript attempt

valueOf() conversion before toString() conversion, except for the Date class,

which performs toString() conversion. Objects that are not part of core Java-

Script may convert themselves to primitive values in an implementation-defined

way.

— Any other combinations of values are not equal.

# Comparison operators

1 + 2 // Addition. Result is 3.

"1" + "2" // Concatenation. Result is "12".

"1" + 2 // Concatenation. 2 is converted to "2". Result is "12".

11 < 3 // Numeric comparison. Result is false.

"11" < "3" // String comparison. Result is true.

"11" < 3 // Numeric comparison. "11" converted to 11. Result is false.

"one" < 3 // Numeric comparison. "one" converted to NaN. Result is false.

# Operator IN

The in operator expects a left-side operand that is or can be converted to a string. It

expects a right-side operand that is an object. It evaluates to true if the left-side value

is the name of a property of the right-side object. For example:

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

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

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

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

var data = [7,8,9]; // An array with elements 0, 1, and 2

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

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

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

# InstanceOf

var d = new Date(); // Create a new object with the Date() constructor

d instanceof Date; // Evaluates to true; d was created with Date()

d instanceof Object; // Evaluates to true; all objects are instances of Object

d instanceof Number; // Evaluates to false; d is not a Number object

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

a instanceof Array; // Evaluates to true; a is an array

a instanceof Object; // Evaluates to true; all arrays are objects

a instanceof RegExp; // Evaluates to false; arrays are not regular expressions

# &&

var o = { x : 1 };

var p = null;

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

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

For example,

the following two lines of JavaScript code have equivalent effects:

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

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

// These two equalities hold for any values of p and q

!(p && q) === !p || !q

!(p || q) === !p && !q

# Typeof operator

x typeof x

undefined "undefined"

null "object"

true or false "boolean"

any number or NaN "number"

any string "string"

82 | Chapter 4: Expressions and Operators

x typeof x

any function "function"

any nonfunction native object "object"

any host object An implementation-defined string, but not “undefined”, “boolean”, “number”, or “string”.

# Delete Operator

delete is a unary operator that attempts to delete the object property or array element

specified as its operand.1 Like the assignment, increment, and decrement operators,

delete is typically used for its property deletion side effect, and not for the value it

returns. Some examples:

var o = { x: 1, y: 2}; // Start with an object

delete o.x; // Delete one of its properties

"x" in o // => false: the property does not exist anymore

var a = [1,2,3]; // Start with an array

delete a[2]; // Delete the last element of the array

a.length // => 2: array only has two elements now