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

# Eval

**JavaScript does this with the global function eval(): eval("3+2") // => 5**

**If it calls eval("x=1"), it changes the value of the local variable. And if the function calls eval("var y = 3;"), it has declared a new local variable y. Similarly a function can declare a local function with code like this: eval("function f() { return x+1; }"); If you call eval() from top-level code, it operates on global variables and global functions, of course.**

# **The Conditional Operator (?:)**

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

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

The delete Operator

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

Here are some example uses of the delete operator:

var o = {x:1, y:2}; // Define a variable; initialize it to an object

delete o.x; // Delete one of the object properties; returns true

typeof o.x; // Property does not exist; returns "undefined"

delete o.x; // Delete a nonexistent property; returns true

delete o; // Can't delete a declared variable; returns false.

// Would raise an exception in strict mode.

delete 1; // Argument is not an lvalue: returns true

this.x = 1; // Define a property of the a global object without var

delete x; // Try to delete it: returns true in non-strict mode

The Comma Operator (,)

for(var i=0,j=10; i < j; i++,j--)

console.log(i+j);

Dat loop

Consider the following for loop (for loops will be covered in §5.5.3):

// Initialize an array a

for(i = 0; i < a.length; a[i++] = 0) ;

Switch

The following switch statement is equivalent to the repeated if/else statements

shown in the previous section:

switch(n) {

case 1: // Start here if n == 1

// Execute code block #1.

break;

// Stop here

case 2: // Start here if n == 2

// Execute code block #2.

break; // Stop here

case 3: // Start here if n == 3

// Execute code block #3.

break; // Stop here

default: // If all else fails...

// Execute code block #4.

break; // stop here

}

Here is a more realistic example of the switch statement; it converts a value to a string

in a way that depends on the type of the value:

function convert(x) {

switch(typeof x) {

case 'number': // Convert the number to a hexadecimal integer

return x.toString(16);

case 'string': // Return the string enclosed in quotes

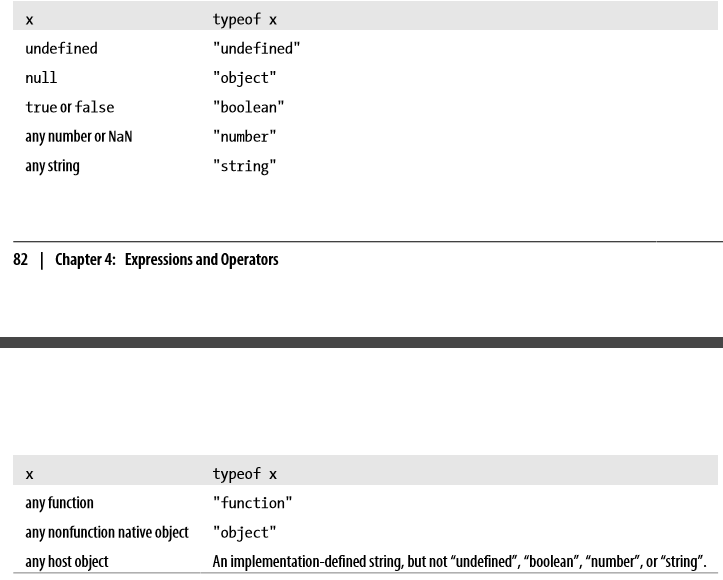
return '"' + x + '"';

default: // Convert any other type in the usual way

return String(x);

}

}



**You might use the typeof operator in an expression like this: (typeof value == "string") ? "'" + value + "'" : value**

**Do While**

do/while loop:

function printArray(a) {

var len = a.length, i = 0;

if (len == 0)

console.log("Empty Array");

else {

do {

console.log(a[i]);

} while (++i < len);

}

}

LinkedList Traversing

The following code uses a for loop to traverse a linked list

data structure and return the last object in the list (i.e., the first object that does not

have a next property):

function tail(o) { // Return the tail of linked list o

for(; o.next; o = o.next) /\* empty \*/ ; // Traverse while o.next is truthy

return o;

}

For In

It is easy to use a regular for loop to iterate through the elements of an array:

for(var i = 0; i < a.length; i++) // Assign array indexes to variable i

console.log(a[i]); // Print the value of each array element

The for/in loop makes it easy to do the same for the properties of an object:

for(var p in o) // Assign property names of o to variable p

console.log(o[p]); // Print the value of each property

Throw

function factorial(x) {

// If the input argument is invalid, throw an exception!

if (x < 0) throw new Error("x must not be negative");

// Otherwise, compute a value and return normally

for(var f = 1; x > 1; f \*= x, x--) /\* empty \*/ ;

return f;

}

When an exception is thrown, the JavaScript interpreter immediately stops normal

program execution and jumps to the nearest exception handler. Exception handlers are

written using the catch clause of the try/catch/finally statement,

Try Catch

try {

// Ask the user to enter a number

var n = Number(prompt("Please enter a positive integer", ""));

// Compute the factorial of the number, assuming the input is valid

var f = factorial(n);

// Display the result

alert(n + "! = " + f);

}

catch (ex) { // If the user's input was not valid, we end up here

alert(ex); // Tell the user what the error is

}

With

If you need to write expressions like this a number of times, you can use the with

statement to add the form object to the scope chain:

with(document.forms[0]) {

// Access form elements directly here. For example:

name.value = "";

address.value = "";

email.value = "";

}

Debugger

To help you in debugging this

problem, you might alter f() so that it begins like this:

function f(o) {

if (o === undefined) debugger; // Temporary line for debugging purposes

... // The rest of the function goes here.

}

Now, when f() is called with no argument, execution will stop, and you can use the

debugger to inspect the call stack and find out where this incorrect call is coming from.

Mutable

Recall from §3.7 that objects are *mutable* and are manipulated by reference rather than

by value. If the variable x refers to an object, and the code var y = x; is executed, the

variable y holds a reference to the same object, not a copy of that object. Any modifications

made to the object through the variable y are also visible through the variable x.

Three object attributes

In addition to its properties, every object has three associated *object attributes*:

• An object’s *prototype* is a reference to another object from which properties are

inherited.

• An object’s *class* is a string that categorizes the type of an object.

• An object’s *extensible* flag specifies (in ECMAScript 5) whether new properties may

be added to the object.

Objects

Finally, here are some terms we’ll use to distinguish among three broad categories of

JavaScript objects and two types of properties:

• A *native object* is an object or class of objects defined by the ECMAScript specification.

Arrays, functions, dates, and regular expressions (for example) are native

objects.

• A *host object* is an object defined by the host environment (such as a web browser)

within which the JavaScript interpreter is embedded. The HTMLElement objects

that represent the structure of a web page in client-side JavaScript are host objects.

Host objects may also be native objects, as when the host environment defines

methods that are normal JavaScript Function objects.

• A *user-defined* object is any object created by the execution of JavaScript code.

• An *own property* is a property defined directly on an object.

• An *inherited property* is a property defined by an object’s prototype object.

Creating Objects

Objects can be created with object literals, with the new keyword, and (in

ECMAScript 5) with the Object.create() function. The subsections below describe

each technique.

Here are some examples:

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

var point = { x:0, y:0 }; // Two properties

var point2 = { x:point.x, y:point.y+1 }; // More complex values

var book = {

"main title": "JavaScript", // Property names include spaces,

'sub-title': "The Definitive Guide", // and hyphens, so use string literals

"for": "all audiences", // for is a reserved word, so quote

author: { // The value of this property is

firstname: "David", // itself an object. Note that

surname: "Flanagan" // these property names are unquoted.

}

};

Core JavaScript includes built-in constructors for

native types. For example:

var o = new Object(); // Create an empty object: same as {}.

var a = new Array(); // Create an empty array: same as [].

var d = new Date(); // Create a Date object representing the current

Object.create() is a static function, not a method invoked on individual objects. To

use it, simply pass the desired prototype object:

var o1 = Object.create({x:1, y:2}); // o1 inherits properties x and y.

If you want to create an ordinary empty object (like the object returned by {} or new

Object()), pass Object.prototype:

var o3 = Object.create(Object.prototype); // o3 is like {} or new Object().

Prototypes

All objects created by object literals have the same prototype object, and we can refer

to this prototype object in JavaScript code as Object.prototype. Objects created using

the new keyword and a constructor invocation use the value of the prototype property

of the constructor function as their prototype. So the object created by new Object()

inherits from Object.prototype just as the object created by {} does. Similarly, the

object created by new Array() uses Array.prototype as its prototype, and the object

created by new Date() uses Date.prototype as its prototype.

Objects As Associative Arrays

object.property

object["property"]

var addr = "";

for(i = 0; i < 4; i++) {

addr += customer["address" + i] + '\n';

This code reads and concatenates the address0, address1, address2, and address3

properties of the customer object.

Inheritance

This continues until the property x is

found or until an object with a null prototype is searched. As you can see, the *prototype*

attribute of an object creates a chain or linked list from which properties are

inherited.

var o = {} // o inherits object methods from Object.prototype

o.x = 1; // and has an own property x.

var p = inherit(o); // p inherits properties from o and Object.prototype

p.y = 2; // and has an own property y.

var q = inherit(p); // q inherits properties from p, o, and Object.prototype

q.z = 3; // and has an own property z.

var s = q.toString(); // toString is inherited from Object.prototype

q.x + q.y // => 3: x and y are inherited from o and p

The fact that inheritance occurs when

querying properties but not when setting them is a key feature of JavaScript because it

allows us to selectively override inherited properties:

var unitcircle = { r:1 }; // An object to inherit from

var c = inherit(unitcircle); // c inherits the property r

c.x = 1; c.y = 1; // c defines two properties of its own

c.r = 2; // c overrides its inherited property

unitcircle.r; // => 1: the prototype object is not affected

Property Access Errors

// A verbose and explicit technique

var len = undefined;

if (book) {

if (book.subtitle) len = book.subtitle.length;

}

// A concise and idiomatic alternative to get subtitle length or undefined

var len = book && book.subtitle && book.subtitle.length;

An attempt to set a property p of an object o

fails in these circumstances:

• o has an own property p that is read-only: it is not possible to set read-only properties.

(See the defineProperty() method, however, for an exception that allows

configurable read-only properties to be set.)

• o has an inherited property p that is read-only: it is not possible to hide an inherited

read-only property with an own property of the same name.

• o does not have an own property p; o does not inherit a property p with a setter

method, and o’s *extensible* attribute (see §6.8.3) is false. If p does not already

exist on o, and if there is no setter method to call, then p must be added

Deleting Properties

delete also evaluates to true when used

(meaninglessly) with an expression that is not a property access expression:

o = {x:1}; // o has own property x and inherits property toString

delete o.x; // Delete x, and return true

delete o.x; // Do nothing (x doesn't exist), and return true

delete o.toString; // Do nothing (toString isn't an own property), return true

delete 1; // Nonsense, but evaluates to true

delete does not remove properties that have a *configurable* attribute of false

In non-strict mode (and in

ECMAScript 3), delete simply evaluates to false in this case:

delete Object.prototype; // Can't delete; property is non-configurable

var x = 1; // Declare a global variable

delete this.x; // Can't delete this property

function f() {} // Declare a global function

delete this.f; // Can't delete this property either

When deleting configurable properties of the global object in non-strict mode, you can

omit the reference to the global object and simply follow the delete operator with the

property name:

this.x = 1; // Create a configurable global property (no var)

delete x; // And delete it

In strict mode, however, delete raises a SyntaxError if its operand is an unqualified

identifier like x, and you have to be explicit about the property access:

delete x; // SyntaxError in strict mode

delete this.x; // This works

Testing Properties

var o = { x: 1 }

"x" in o; // true: o has an own property "x"

"y" in o; // false: o doesn't have a property "y"

"toString" in o; // true: o inherits a toString property

The hasOwnProperty() method of an object tests whether that object has an own property

with the given name. It returns false for inherited properties:

var o = { x: 1 }

o.hasOwnProperty("x"); // true: o has an own property x

o.hasOwnProperty("y"); // false: o doesn't have a property y

o.hasOwnProperty("toString"); // false: toString is an inherited property

The propertyIsEnumerable() refines the hasOwnProperty() test. It returns true only if

the named property is an own property and its *enumerable* attribute is true. Certain

built-in properties are not enumerable. Properties created by normal JavaScript code

are enumerable unless you’ve used one of the ECMAScript 5 methods shown later to

make them nonenumerable.

var o = inherit({ y: 2 });

o.x = 1;

o.propertyIsEnumerable("x"); // true: o has an own enumerable property x

o.propertyIsEnumerable("y"); // false: y is inherited, not own

Object.prototype.propertyIsEnumerable("toString"); // false: not enumerable

Instead of using the in operator it is often sufficient to simply query the property and

use !== to make sure it is not undefined:

var o = { x: 1 }

o.x !== undefined; // true: o has a property x

o.y !== undefined; // false: o doesn't have a property y

o.toString !== undefined; // true: o inherits a toString property

There is one thing the in operator can do that the simple property access technique

shown above cannot do. in can distinguish between properties that do not exist and

properties that exist but have been set to undefined. Consider this code:

var o = { x: undefined } // Property is explicitly set to undefined

o.x !== undefined // false: property exists but is undefined

o.y !== undefined // false: property doesn't even exist

"x" in o // true: the property exists

"y" in o // false: the property doesn't exists

delete o.x; // Delete the property x

"x" in o // false: it doesn't exist anymore

Enumerating Properties

var o = {x:1, y:2, z:3}; // Three enumerable own properties

o.propertyIsEnumerable("toString") // => false: not enumerable

for(p in o) // Loop through the properties

console.log(p); // Prints x, y, and z, but not toString

To guard against this, you might want to filter the properties returned

by for/in. Here are two ways you might do so:

for(p in o) {

if (!o.hasOwnProperty(p)) continue; // Skip inherited properties

for(p in o) {

if (typeof o[p] === "function") continue; // Skip methods

}

In addition to the for/in loop, ECMAScript 5 defines two functions that enumerate

property names. The first is Object.keys(), which returns an array of the names of the

enumerable own properties of an object. It works just like the keys() utility function

shown in Example 6-2.

The second ECMAScript 5 property enumeration function is Object.getOwnProperty

Names(). It works like Object.keys() but returns the names of all the own properties of

the specified object, not just the enumerable properties. There is no way to write this

function in ECMAScript 3, because ECMAScript 3 does not provide a way to obtain

the nonenumerable properties of an object.

Getters and Setters

The easiest way to define accessor properties is with an extension to the object literal

syntax:

var o = {

// An ordinary data property

data\_prop: value,

// An accessor property defined as a pair of functions

get accessor\_prop() { /\* function body here \*/ },

set accessor\_prop(value) { /\* function body here \*/ }

};

var p = {

// x and y are regular read-write data properties.

x: 1.0,

y: 1.0,

// r is a read-write accessor property with getter and setter.

// Don't forget to put a comma after accessor methods.

get r() { return Math.sqrt(this.x\*this.x + this.y\*this.y); },

set r(newvalue) {

var oldvalue = Math.sqrt(this.x\*this.x + this.y\*this.y);

var ratio = newvalue/oldvalue;

this.x \*= ratio;

this.y \*= ratio;

},

// theta is a read-only accessor property with getter only.

get theta() { return Math.atan2(this.y, this.x); }

};

Accessor properties are inherited, just as data properties are, so you can use the object

p defined above as a prototype for other points. You can give the new objects their own

x and y properties, and they’ll inherit the r and theta properties:

var q = inherit(p); // Create a new object that inherits getters and setters

q.x = 0, q.y = 0; // Create q's own data properties

console.log(q.r); // And use the inherited accessor properties

console.log(q.theta);

Property Attributes

The ECMAScript 5 methods for querying and setting the attributes of a property use

an object called a *property descriptor* to represent the set of four attributes.

To obtain the property descriptor for a named property of a specified object, call

Object.getOwnPropertyDescriptor():

// Returns {value: 1, writable:true, enumerable:true, configurable:true}

Object.getOwnPropertyDescriptor({x:1}, "x");

// Now query the octet property of the random object defined above.

// Returns { get: /\*func\*/, set:undefined, enumerable:true, configurable:true}

Object.getOwnPropertyDescriptor(random, "octet");

// Returns undefined for inherited properties and properties that don't exist.

Object.getOwnPropertyDescriptor({}, "x"); // undefined, no such prop

Object.getOwnPropertyDescriptor({}, "toString"); // undefined, inherited

To set the attributes of a property, or to create a new property with the specified attributes,

call Object.defineProperty(), passing the object to be modified, the name of

the property to be created or altered, and the property descriptor object:

var o = {}; // Start with no properties at all

// Add a nonenumerable data property x with value 1.

Object.defineProperty(o, "x", { value : 1,

writable: true,

enumerable: false,

configurable: true});

// Check that the property is there but is nonenumerable

o.x; // => 1

Object.keys(o) // => []

// Now modify the property x so that it is read-only

Object.defineProperty(o, "x", { writable: false });

// Try to change the value of the property

o.x = 2; // Fails silently or throws TypeError in strict mode

o.x // => 1

// The property is still config

urable, so we can change its value like this:

Object.defineProperty(o, "x", { value: 2 });

o.x // => 2

// Now change x from a data property to an accessor property

Object.defineProperty(o, "x", { get: function() { return 0; } });

o.x // => 0

The property descriptor you

To set the attributes of a property, or to create a new property with the specified attributes,

call Object.defineProperty(), passing the object to be modified, the name of

the property to be created or altered, and the property descriptor object:

var o = {}; // Start with no properties at all

// Add a nonenumerable data property x with value 1.

Object.defineProperty(o, "x", { value : 1,

writable: true,

enumerable: false,

configurable: true});

// Check that the property is there but is nonenumerable

o.x; // => 1

Object.keys(o) // => []

// Now modify the property x so that it is read-only

Object.defineProperty(o, "x", { writable: false });

// Try to change the value of the property

o.x = 2; // Fails silently or throws TypeError in strict mode

o.x // => 1

// The property is still configurable, so we can change its value like this:

Object.defineProperty(o, "x", { value: 2 });

o.x // => 2

// Now change x from a data property to an accessor property

Object.defineProperty(o, "x", { get: function() { return 0; } });

o.x // => 0

The property descriptor you

The class Attribute

function classof(o) {

if (o === null) return "Null";

if (o === undefined) return "Undefined";

return Object.prototype.toString.call(o).slice(8,-1);

}

This classof() function works for any JavaScript value. Numbers, strings, and booleans

behave like objects when the toString() method is invoked on them

The extensible Attribute

ECMAScript 5 defines functions for querying and setting the extensibility of an object.

To determine whether an object is extensible, pass it to Object.isExtensible(). To

make an object nonextensible, pass it to Object.preventExtensions().

Serializing Objects

Object *serialization* is the process of converting an object’s state to a string from which

it can later be restored. ECMAScript 5 provides native functions JSON.stringify() and

JSON.parse() to serialize and restore JavaScript objects. These functions use the JSON

data interchange format. JSON stands for “JavaScript Object Notation,” and its syntax

is very similar to that of JavaScript object and array literals:

o = {x:1, y:{z:[false,null,""]}}; // Define a test object

s = JSON.stringify(o); // s is '{"x":1,"y":{"z":[false,null,""]}}'

p = JSON.parse(s); // p is a deep copy of o

JSON.stringify() serializes only the enumerable own properties of an

object. If a property value cannot be serialized, that property is simply omitted from

the stringified output.

Creating Arrays

The easiest way to create an array is with an array literal, which is simply a commaseparated

list of array elements within square brackets. For example:

141

var empty = []; // An array with no elements

var primes = [2, 3, 5, 7, 11]; // An array with 5 numeric elements

var misc = [ 1.1, true, "a", ]; // 3 elements of various types + trailing comma

The values in an array literal need not be constants; they may be arbitrary expressions:

var base = 1024;

var table = [base, base+1, base+2, base+3];

Array literals can contain object literals or other array literals:

var b = [[1,{x:1, y:2}], [2, {x:3, y:4}]];

If you omit a value from an array literal, the omitted element is given the value

undefined:

var count = [1,,3]; // An array with 3 elements, the middle one undefined.

var undefs = [,,]; // An array with 2 elements, both undefined.

Another way to create an array is with the Array() constructor. You can invoke this

constructor in three distinct ways:

• Call it with no arguments:

var a = new Array();

This method creates an empty array with no elements and is equivalent to the array

literal [].

• Call it with a single numeric argument, which specifies a length:

var a = new Array(10);

This technique creates an array with the specified length. This form of the

Array() constructor can be used to preallocate an array when you know in advance

how many elements will be required. Note that no values are stored in the array,

and the array index properties “0”, “1”, and so on are not even defined for the array.

• Explicitly specify two or more array elements or a single non-numeric element for

the array:

var a = new Array(5, 4, 3, 2, 1, "testing, testing");

Above, for example, we created an array a with a single element. We then

assigned values at indexes 1, 2, and 3. The length property of the array changed as we

did so:

a.length // => 4

Also, if you index an array with a string that happens

to be a non-negative integer, it behaves as an array index, not an object property.

The same is true if you use a floating-point number that is the same as an integer:

a[-1.23] = true; // This creates a property named "-1.23"

a["1000"] = 0; // This the 1001st element of the array

a[1.000] // Array index 1. Same as a[1]

When you try to

query a nonexistent property of any object, you don’t get an error, you simply get

undefined. This is just as true for arrays as it is for objects:

a = [true, false]; // This array has elements at indexes 0 and 1

a[2] // => undefined. No element at this index.

a[-1] // => undefined. No property with this name.

Sparse Arraysv

A *sparse* array is one in which the elements do not have contiguous indexes starting at

0. Normally, the length property of an array specifies the number of elements in the

array. If the array is sparse, the value of the length property is greater than the number

of elements. Sparse arrays can be created with the Array() constructor or simply by

assigning to an array index larger than the current array length.

a = new Array(5); // No elements, but a.length is 5.

a = []; // Create an array with no elements and length = 0.

a[1000] = 0; // Assignment adds one element but sets length to 1001.

Note that when you omit value in an array literal, you are not creating a sparse array.

The omitted element exists in the array and has the value undefined. This is subtly

different than array elements that do not exist at all. You can detect the difference

between these two cases with the in operator:

var a1 = [,,,]; // This array is [undefined, undefined, undefined]

var a2 = new Array(3); // This array has no values at all

0 in a1 // => true: a1 has an element with index 0

0 in a2 // => false: a2 has no element with index 0

Array Length

Its value is one more than the

highest index in the array:

[].length // => 0: the array has no elements

['a','b','c'].length // => 3: highest index is 2, length is 3

The second special behavior that arrays implement in order to maintain the length

invariant is that if you set the length property to a non-negative integer n smaller than

its current value, any array elements whose index is greater than or equal to n are deleted

from the array:

a = [1,2,3,4,5]; // Start with a 5-element array.

a.length = 3; // a is now [1,2,3].

a.length = 0; // Delete all elements. a is [].

a.length = 5; // Length is 5, but no elements, like new Array(5)

In ECMAScript 5, you can make the length property of an array read-only with

Object.defineProperty() (see §6.7):

a = [1,2,3]; // Start with a 3-element array.

Object.defineProperty(a, "length", // Make the length property

{writable: false}); // readonly.

a.length = 0;

Adding and Deleting Array Elements

We’ve already seen the simplest way to add elements to an array: just assign values to

new indexes:

a = [] // Start with an empty array.

a[0] = "zero"; // And add elements to it.

a[1] = "one";

You can also use the push() method to add one or more values to the end of an array:

7.5 Adding and Deleting Array Elements | 145

**Core JavaScript**

a = []; // Start with an empty array

a.push("zero") // Add a value at the end. a = ["zero"]

a.push("one", "two") // Add two more values. a = ["zero", "one", "two"]

You can delete array elements with the delete operator, just as you can delete object

properties:

a = [1,2,3];

delete a[1]; // a now has no element at index 1

1 in a // => false: no array index 1 is defined

a.length // => 3: delete does not affect array length

Iterating Arrays

var keys = Object.keys(o); // Get an array of property names for object o

var values = [] // Store matching property values in this array

for(var i = 0; i < keys.length; i++) { // For each index in the array

var key = keys[i]; // Get the key at that index

values[i] = o[key]; // Store the value in the values array

}

In nested loops, or other contexts where performance is critical, you may sometimes

see this basic array iteration loop optimized so that the array length is only looked up

once rather than on each iteration:

These examples assume that the array is dense and that all elements contain valid data.

If this is not the case, you should test the array elements before using them. If you want

to exclude null, undefined, and nonexistent elements, you can write this:

for(var i = 0; i < a.length; i++) {

if (!a[i]) continue; // Skip null, undefined, and nonexistent elements

// loop body here

}

If you only want to skip undefined and nonexistent elements, you might write:

for(var i = 0; i < a.length; i++) {

if (a[i] === undefined) continue; // Skip undefined + nonexistent elements

// loop body here

}

Finally, if you only want to skip indexes for which no array element exists but still want

to handle existing undefined elements, do this:

for(var i = 0; i < a.length; i++) {

if (!(i in a)) continue ; // Skip nonexistent elements

// loop body here

}

As noted in §6.5, a for/in loop can return the names of inherited properties, such as

the names of methods that have been added to Array.prototype. For this reason you

should not use a for/in loop on an array unless you include an additional test to filter

out unwanted properties. You might use either of these tests:

for(var i in a) {

if (!a.hasOwnProperty(i)) continue; // Skip inherited properties

// loop body here

}

for(var i in a) {

// Skip i if it is not a non-negative integer

if (String(Math.floor(Math.abs(Number(i)))) !== i) continue;

}

ECMAScript 5 defines a number of new methods for iterating array elements by passing

each one, in index order, to a function that you define. The forEach() method is the

most general of these methods:

var data = [1,2,3,4,5]; // This is the array we want to iterate

var sumOfSquares = 0; // We want to compute the sum of the squares of data

data.forEach(function(x) { // Pass each element of data to this function

sumOfSquares += x\*x; // add up the squares

});

sumOfSquares // =>55 : 1+4+9+16+25

Multidimensional Arrays

Here is a concrete example that uses

a two-dimensional array as a multiplication table:

// Create a multidimensional array

var table = new Array(10); // 10 rows of the table

for(var i = 0; i < table.length; i++)

table[i] = new Array(10); // Each row has 10 columns

// Initialize the array

for(var row = 0; row < table.length; row++) {

for(col = 0; col < table[row].length; col++) {

table[row][col] = row\*col;

}

}

// Use the multidimensional array to compute 5\*7

var product = table[5][7]; // 35

Array Methods

join()

The Array.join() method converts all the elements of an array to strings and concatenates

them, returning the resulting string. You can specify an optional string that

separates the elements in the resulting string. If no separator string is specified, a comma

is used. For example, the following lines of code produce the string “1,2,3”:

var a = [1, 2, 3]; // Create a new array with these three elements

a.join(); // => "1,2,3"

a.join(" "); // => "1 2 3"

a.join(""); // => "123"

var b = new Array(10); // An array of length 10 with no elements

b.join('-') // => '---------': a string of 9 hyphens

reverse()

var a = [1,2,3];

a.reverse().join() // => "3,2,1" and a is now [3,2,1]

sort()

Array.sort() sorts the elements of an array in place and returns the sorted array. When

sort() is called with no arguments, it sorts the array elements in alphabetical order

(temporarily converting them to strings to perform the comparison, if necessary):

var a = new Array("banana", "cherry", "apple");

a.sort();

var s = a.join(", "); // s == "apple, banana, cherry"

If an array contains undefined elements, they are sorted to the end of the array.

var a = [33, 4, 1111, 222];

a.sort(); // Alphabetical order: 1111, 222, 33, 4

a.sort(function(a,b) { // Numerical order: 4, 33, 222, 1111

7.8 Array Methods | 149

**Core JavaScript**

return a-b; // Returns &lt; 0, 0, or &gt; 0, depending on order

});

a.sort(function(a,b) {return b-a}); // Reverse numerical order

As another example of sorting array items, you might perform a case-insensitive alphabetical

sort on an array of strings by passing a comparison function that converts

both of its arguments to lowercase (with the toLowerCase() method) before comparing

them:

a = ['ant', 'Bug', 'cat', 'Dog']

a.sort(); // case-sensitive sort: ['Bug','Dog','ant',cat']

a.sort(function(s,t) { // Case-insensitive sort

var a = s.toLowerCase();

var b = t.toLowerCase();

if (a < b) return -1;

if (a > b) return 1;

return 0;

}); // => ['ant','Bug','cat','Dog']

concat()

var a = [1,2,3];

a.concat(4, 5) // Returns [1,2,3,4,5]

a.concat([4,5]); // Returns [1,2,3,4,5]

a.concat([4,5],[6,7]) // Returns [1,2,3,4,5,6,7]

a.concat(4, [5,[6,7]]) // Returns [1,2,3,4,5,[6,7]]

slice()

var a = [1,2,3,4,5];

a.slice(0,3); // Returns [1,2,3]

a.slice(3); // Returns [4,5]

a.slice(1,-1); // Returns [2,3,4]

a.slice(-3,-2); // Returns [3]

splice()

var a = [1,2,3,4,5,6,7,8];

a.splice(4); // Returns [5,6,7,8]; a is [1,2,3,4]

a.splice(1,2); // Returns [2,3]; a is [1,4]

a.splice(1,1); // Returns [4]; a is [1]

The first two arguments to splice() specify which array elements are to be deleted.

These arguments may be followed by any number of additional arguments that specify

elements to be inserted into the array, starting at the position specified by the first

argument. For example:

var a = [1,2,3,4,5];

a.splice(2,0,'a','b'); // Returns []; a is [1,2,'a','b',3,4,5]

a.splice(2,2,[1,2],3); // Returns ['a','b']; a is [1,2,[1,2],3,3,4,5]

push() and pop()

var stack = []; // stack: []

stack.push(1,2); // stack: [1,2] Returns 2

stack.pop(); // stack: [1] Returns 2

stack.push(3); // stack: [1,3] Returns 2

stack.pop(); // stack: [1] Returns 3

stack.push([4,5]); // stack: [1,[4,5]] Returns 2

stack.pop() // stack: [1] Returns [4,5]

stack.pop(); // stack: [] Returns 1

unshift() and shift()

var a = []; // a:[]

a.unshift(1); // a:[1] Returns: 1

a.unshift(22); // a:[22,1] Returns: 2

a.shift(); // a:[1] Returns: 22

a.unshift(3,[4,5]); // a:[3,[4,5],1] Returns: 3

a.shift(); // a:[[4,5],1] Returns: 3

a.shift(); // a:[1] Returns: [4,5]

a.shift(); // a:[] Returns: 1

forEach()

var data = [1,2,3,4,5]; // An array to sum

// Compute the sum of the array elements

var sum = 0; // Start at 0

data.forEach(function(value) { sum += value; }); // Add each value to sum

sum // => 15

// Now increment each array element

data.forEach(function(v, i, a) { a[i] = v + 1; });

data

map()

a = [1, 2, 3];

b = a.map(function(x) { return x\*x; }); // b is [1, 4, 9]

filter()

a = [5, 4, 3, 2, 1];

smallvalues = a.filter(function(x) { return x < 3 }); // [2, 1]

everyother = a.filter(function(x,i) { return i%2==0 }); // [5, 3, 1]

Note that filter() skips missing elements in sparse arrays, and that its return value is

always dense. To close the gaps in a sparse array, you can do this:

var dense = sparse.filter(function() { return true; });

And to close gaps and remove undefined and null elements you can use filter like this:

a = a.filter(function(x) { return x !== undefined && x != null; });

every() and some()

The every() method is like the mathematical “for all” quantifier ∀: it returns true if

and only if your predicate function returns true for all elements in the array:

154 | Chapter 7: Arrays

a = [1,2,3,4,5];

a.every(function(x) { return x < 10; }) // => true: all values < 10.

a.every(function(x) { return x % 2 === 0; }) // => false: not all values even.

The some() method is like the mathematical “there exists” quantifier ∃: it returns

true if there exists at least one element in the array for which the predicate returns

true, and returns false if and only if the predicate returns false for all elements of

the array:

a = [1,2,3,4,5];

a.some(function(x) { return x%2===0; }) // => true a has some even numbers.

a.some(isNaN)

reduce(), reduceRight()

var a = [1,2,3,4,5]

var sum = a.reduce(function(x,y) { return x+y }, 0); // Sum of values

var product = a.reduce(function(x,y) { return x\*y }, 1); // Product of values

var max = a.reduce(function(x,y) { return (x>y)?x:y; }); // Largest value

indexOf() and lastIndexOf()

a = [0,1,2,1,0];

a.indexOf(1) // => 1: a[1] is 1

a.lastIndexOf(1) // => 3: a[3] is 1

a.indexOf(3) // => -1: no element has value 3

This demonstrates how the second argument to indexOf() can

be used to find matches beyond the first.

// Find all occurrences of a value x in an array a and return an array

// of matching indexes

function findall(a, x) {

var results = [], // The array of indexes we'll return

len = a.length, // The length of the array to be searched

pos = 0; // The position to search from

while(pos < len) { // While more elements to search...

pos = a.indexOf(x, pos); // Search

if (pos === -1) break; // If nothing found, we're done.

results.push(pos); // Otherwise, store index in array

pos = pos + 1; // And start next search at next element

}

return results; // Return array of indexes

}

Note that strings

Array Type

Given an unknown object, it is often useful to be able to determine whether it is an

array or not. In ECMAScript 5, you can do this with the Array.isArray() function:

Array.isArray([]) // => true

Array.isArray({}) // => false

Array-Like Objects

As we’ve seen, JavaScript arrays have some special features that other objects do

not have:

• The length property is automatically updated as new elements are added to the list.

• Setting length to a smaller value truncates the array.

• Arrays inherit useful methods from Array.prototype.

• Arrays have a *class* attribute of “Array”.

The following code takes a regular object, adds properties to make it an array-like

object, and then iterates through the “elements” of the resulting pseudo-array:

var a = {}; // Start with a regular empty object

// Add properties to make it "array-like"

var i = 0;

while(i < 10) {

a[i] = i \* i;

i++;

}

a.length = i;

// Now iterate through it as if it were a real array

var total = 0;

for(var j = 0; j < a.length; j++)

total += a[j];

Strings As Arrays

var s = test;

s.charAt(0) // => "t"

s[1] // => "e"

The typeof operator still returns “string” for strings, of course, and the

Array.isArray() method returns false if you pass it a string.

The fact that strings behave like arrays also means, however, that we

can apply generic array methods to them. For example:

s = "JavaScript"

Array.prototype.join.call(s, " ") // => "J a v a S c r i p t"

Array.prototype.filter.call(s, // Filter the characters of the string

function(x) {

return x.match(/[^aeiou]/); // Only match nonvowels

}).join("") // => "JvScrpt"

Keep in mind that strings are immutable values, so when they are treated as arrays, they

are read-only arrays. Array methods like push(), sort(), reverse(), and splice() modify

an array in place and do not work on strings.

Defining Functions

function factorial(x) {

if (x <= 1) return 1;

return x \* factorial(x-1);

}

var f = function fact(x) { if (x <= 1) return 1; else return x\*fact(x-1); };

// Function expressions can also be used as arguments to other functions:

data.sort(function(a,b) { return a-b; });

// Function expressions are sometimes defined and immediately invoked:

var tensquared = (function(x) {return x\*x;}(10));

Nested Functions

function hypotenuse(a, b) {

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

return Math.sqrt(square(a) + square(b));

}

The interesting thing about nested functions is their variable scoping rules: they can

access the parameters and variables of the function (or functions) they are nested within.

In the code above, for example, the inner function square() can read and write the

parameters a and b defined by the outer function hypotenuse(). These scope rules for

nested functions are very important, and we’ll consider them again in

Invoking Functions

The JavaScript code that makes up the body of a function is not executed when the

function is defined but when it is invoked. JavaScript functions can be invoked in four

ways:

• as functions,

• as methods,

• as constructors, and

• indirectly through their call() and apply() methods.

Function Invocation

The following code includes a number of regular function invocation

expressions:

printprops({x:1});

var total = distance(0,0,2,1) + distance(2,1,3,5);

var probability = factorial(5)/factorial(13);

Method Invocation

If you have a function f and an object o, you can define a method named m of

o with the following line:

o.m = f;

Having defined the method

var calculator = { // An object literal

operand1: 1,

operand2: 1,

add: function() {

// Note the use of the this keyword to refer to this object.

this.result = this.operand1 + this.operand2;

}

};

calculator.add(); // A method invocation to compute 1+1.

calculator.result // => 2

The following are

both method invocations, for example:

o["m"](x,y); // Another way to write o.m(x,y).

a[0](z) // Also a method invocation (assuming a[0] is a function).

Method invocations may also involve more complex property access expressions:

customer.surname.toUpperCase(); // Invoke method on customer.surname

f().m(); // Invoke method m() on return value of f()

Method Chaining

When you write a method that does not have a return value of its own, consider having

the method return this. If you do this consistently throughout your API, you will enable

a style of programming known as *method chaining*2 in which an object can be named

once and then multiple methods can be invoked on it:

shape.setX(100).setY(100).setSize(50).setOutline("red").setFill("blue").draw();

Constructor Invocation

If a function or method invocation is preceded by the keyword new, then it is a

constructor invocation.

You can always omit a pair of empty parentheses in a constructor

invocation and the following two lines, for example, are equivalent:

var o = new Object();

var o = new Object;

Indirect Invocation

JavaScript functions are objects and like all JavaScript objects, they have methods. Two

of these methods, call() and apply(), invoke the function indirectly. Both methods

allow you to explicitly specify the this value for the invocation, which means you can

invoke any function as a method of any object, even if it is not actually a method of

that object. Both methods also allow you to specify the arguments for the invocation.

The call() method uses its own argument list as arguments to the function and the

apply() method expects an array of values to be used as arguments.

Optional Parameters

// Append the names of the enumerable properties of object o to the

// array a, and return a. If a is omitted, create and return a new array.

function getPropertyNames(o, /\* optional \*/ a) {

if (a === undefined) a = []; // If undefined, use a new array

for(var property in o) a.push(property);

return a;

}

// This function can be invoked with 1 or 2 arguments:

var a = getPropertyNames(o); // Get o's properties into a new array

getPropertyNames(p,a); // append p's properties to that array

Instead of using an if statement in the first line of this function, you can use the ||

operator in this idiomatic way:

a = a || [];

Note that when designing functions with optional arguments, you should be sure to

put the optional ones at the end of the argument list so that they can be omitted. The

programmer who calls your function cannot omit the first argument and pass the second:

she would have to explicitly pass undefined the first argument.

Variable-Length Argument Lists: The Arguments Object

The Arguments object is useful in a number of ways. The following example shows

how you can use it to verify that a function is invoked with the expected number of

arguments, since JavaScript doesn’t do this for you:

function f(x, y, z)

{

// First, verify that the right number of arguments was passed

if (arguments.length != 3) {

throw new Error("function f called with " + arguments.length +

"arguments, but it expects 3 arguments.");

}

// Now do the actual function...

}

One important use of the Arguments object is to write functions that operate on any

number of arguments. The following function accepts any number of numeric arguments

and returns the value of the largest argument it is passed (see also the built-in

function Math.max(), which behaves the same way):

function max(/\* ... \*/) {

var max = Number.NEGATIVE\_INFINITY;

// Loop through the arguments, looking for, and remembering, the biggest.

for(var i = 0; i < arguments.length; i++)

if (arguments[i] > max) max = arguments[i];

// Return the biggest

return max;

}

var largest = max(1, 10, 100, 2, 3, 1000, 4, 5, 10000, 6); // => 10000

Conversely, changing the value of an

argument through the arguments[] array changes the value that is retrieved by the argument

name. Here is an example that clarifies this:

function f(x) {

console.log(x); // Displays the initial value of the argument

arguments[0] = null; // Changing the array element also changes x!

console.log(x); // Now displays "null"

}

The callee and caller properties

In addition to its array elements, the Arguments object defines callee and caller properties.

In ECMAScript 5 strict mode, these properties are guaranteed to raise a Type-

Error if you try to read or write them.

The caller property gives access to the call stack, and the callee

property is occasionally useful to allow unnamed functions to call themselves

recursively:

var factorial = function(x) {

if (x <= 1) return 1;

return x \* arguments.callee(x-1);

};

Using Object Properties As Arguments

// This version is a little less efficient, but you don't have to

// remember the order of the arguments, and from\_start and to\_start

// default to 0.

function easycopy(args) {

arraycopy(args.from,

args.from\_start || 0, // Note default value provided

args.to,

args.to\_start || 0,

args.length);

}

// Here is how you might invoke easycopy():

var a = [1,2,3,4], b = [];

easycopy({from: a, to: b, length: 4});

Argument Types

JavaScript performs liberal type conversion as needed. So if you

write a function that expects a string argument and then call that function with a value

of some other type, the value you passed will simply be converted to a string when the

function tries to use it as a string. All primitive types can be converted to strings, and

all objects have toString() methods (if not necessarily useful ones), so an error never

occurs in this case.

Note that it uses the isArrayLike() function

from §7.11:

// Return the sum of the elements of array (or array-like object) a.

// The elements of a must all be numbers or null and undefined are ignored.

function sum(a) {

if (isArrayLike(a)) {

var total = 0;

for(var i = 0; i < a.length; i++) { // Loop though all elements

var element = a[i];

if (element == null) continue; // Skip null and undefined

if (isFinite(element)) total += element;

else throw new Error("sum(): elements must be finite numbers");

}

return total;

}

else throw new Error("sum(): argument must be array-like");

}

The following flexisum() method takes this approach (probably to an extreme).

For example, it accepts any number of arguments but recursively processes any

arguments that are arrays. In this way, it can be used as a varargs method or with an

array argument. Furthermore, it tries its best to convert nonnumeric values to numbers

before throwing an error:

function flexisum(a) {

var total = 0;

for(var i = 0; i < arguments.length; i++) {

var element = arguments[i], n;

if (element == null) continue; // Ignore null and undefined arguments

8.3 Function Arguments and Parameters | 175

**Core JavaScript**

if (isArray(element)) // If the argument is an array

n = flexisum.apply(this, element); // compute its sum recursively

else if (typeof element === "function") // Else if it's a function...

n = Number(element()); // invoke it and convert.

else n = Number(element); // Else try to convert it

if (isNaN(n)) // If we couldn't convert to a number, throw an error

throw Error("flexisum(): can't convert " + element + " to number");

total += n; // Otherwise, add n to the total

}

return total;

}

Functions As Values

To understand how functions can be JavaScript data as well as JavaScript syntax, consider

this function definition:

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

This definition creates a new function object and assigns it to the variable square. The

name of a function is really immaterial; it is simply the name of a variable that refers to

the function object. The function can be assigned to another variable and still work the

same way:

var s = square; // Now s refers to the same function that square does

square(4); // => 16

s(4); // => 16

Functions don’t even require names at all, as when they’re assigned to array elements:

var a = [function(x) { return x\*x; }, 20]; // An array literal

a[0](a[1]); // => 400

// We define some simple functions here

function add(x,y) { return x + y; }

function subtract(x,y) { return x - y; }

function multiply(x,y) { return x \* y; }

function divide(x,y) { return x / y; }

// Here's a function that takes one of the above functions

// as an argument and invokes it on two operands

function operate(operator, operand1, operand2) {

return operator(operand1, operand2);

}

// We could invoke this function like this to compute the value (2+3) + (4\*5):

var i = operate(add, operate(add, 2, 3), operate(multiply, 4, 5));

// For the sake of the example, we implement the simple functions again,

// this time using function literals within an object literal;

var operators = {

add: function(x,y) { return x+y; },

subtract: function(x,y) { return x-y; },

multiply: function(x,y) { return x\*y; },

divide: function(x,y) { return x/y; },

pow: Math.pow // Works for predefined functions too

};

// This function takes the name of an operator, looks up that operator

// in the object, and then invokes it on the supplied operands. Note

// the syntax used to invoke the operator function.

function operate2(operation, operand1, operand2) {

if (typeof operators[operation] === "function")

return operators[operation](operand1, operand2);

else throw "unknown operator";

}

// Compute the value ("hello" + " " + "world") like this:

var j = operate2("add", "hello", operate2("add", " ", "world"));

// Using the predefined Math.pow() function:

var k = operate2("pow", 10, 2);

Defining Your Own Function Properties

Here is an example that returns a unique integer whenever it is called:

// Initialize the counter property of the function object.

// Function declarations are hoisted so we really can

// do this assignment before the function declaration.

uniqueInteger.counter = 0;

// This function returns a different integer each time it is called.

// It uses a property of itself to remember the next value to be returned.

function uniqueInteger() {

return uniqueInteger.counter++; // Increment and return counter property

}

As another example, consider the following factorial() function that uses properties

of itself (treating itself as an array) to cache previously computed results:

// Compute factorials and cache results as properties of the function itself.

function factorial(n) {

if (isFinite(n) && n>0 && n==Math.round(n)) { // Finite, positive ints only

if (!(n in factorial)) // If no cached result

factorial[n] = n \* factorial(n-1); // Compute and cache it

return factorial[n]; // Return the cached result

}

else return NaN; // If input was bad

}

factorial[1] = 1; // Initialize the cache to hold this base case.

Functions As Namespaces

function mymodule() {

// Module code goes here.

// Any variables used by the module are local to this function

// instead of cluttering up the global namespace.

}

mymodule(); // But don't forget to invoke the function!

This code defines only a single global variable: the function name “mymodule”. If defining

even a single property is too much, you can define and invoke an anonymous

function in a single expression:

(function() { // mymodule function rewritten as an unnamed expression

// Module code goes here.

}()); // end the function literal and invoke it now.

The open parenthesis before function is required because without it, the Java-

Script interpreter tries to parse the function keyword as a function declaration statement.

With the parenthesis, the interpreter correctly recognizes this as a function

definition expression. It is idiomatic to use the parentheses, even when they are not

required, around a function that is to be invoked immediately after being defined.

Closures

Technically, all JavaScript functions are closures: they are objects, and they have a scope

chain associated with them. Most functions are invoked using the same scope chain

that was in effect when the function was defined, and it doesn’t really matter that there

is a closure involved. Closures become interesting when they are invoked under a different scope chain than the one that was in effect when they were defined. This

happens most commonly when a nested function object is returned from the function

within which it was defined. There are a number of powerful programming techniques

that involve this kind of nested function closures, and their use has become relatively

common in JavaScript programming. Closures may seem confusing when you first encounter

them, but it is important that you understand them well enough to use them

comfortably.

var scope = "global scope"; // A global variable

function checkscope() {

var scope = "local scope"; // A local variable

function f() { return scope; } // Return the value in scope here

return f();

}

checkscope() // => "local scope"

The checkscope() function declares a local variable and then defines and invokes a

function that returns the value of that variable. It should be clear to you why the call

to checkscope() returns “local scope”. Now let’s change the code just slightly. Can you

tell what this code will return?

var scope = "global scope"; // A global variable

function checkscope() {

var scope = "local scope"; // A local variable

function f() { return scope; } // Return the value in scope here

return f;

}

checkscope()() // What does this return?

In this code, a pair of parentheses has moved from inside checkscope() to outside of it.

Instead of invoking the nested function and returning its result, checkscope() now just

returns the nested function object itself. What happens when we invoke that nested

function (with the second pair of parentheses in the last line of code) outside of the

function in which it was defined?

Remember the fundamental rule of lexical scoping: JavaScript functions are executed

using the scope chain that was in effect when they were defined. The nested function

f() was defined under a scope chain in which the variable scope was bound to the value

“local scope”. That binding is still in effect when f is executed, wherever it is executed

from. So the last line of code above returns “local scope”, not “global scope”. This, in

a nutshell, is the surprising and powerful nature of closures: they capture the local

variable (and parameter) bindings of the outer function within which they are defined.

But remember our definition of scope chain from §3.10.3. We described it as a list of

objects, not a stack of bindings. Each time a JavaScript function is invoked, a new object

is created to hold the local variables for that invocation, and that object is added to the

scope chain. When the function returns, that variable binding object is removed from

the scope chain. If there were no nested functions, there are no more references to the

binding object and it gets garbage collected. If there were nested functions defined,

then each of those functions has a reference to the scope chain, and that scope chain

refers to the variable binding object. If those nested functions objects remained within

their outer function, however, then they themselves will be garbage collected, along

with the variable binding object they referred to. But if the function defines a nested

function and returns it or stores it into a property somewhere, then there will be an

external reference to the nested function. It won’t be garbage collected, and the variable

binding object it refers to won’t be garbage collected either.

Private variables like counter need not be exclusive to a single closure: it is perfectly

possible for two or more nested functions to be defined within the same outer function

and share the same scope chain. Consider the following code:

function counter() {

var n = 0;

return {

count: function() { return n++; },

reset: function() { n = 0; }

};

}

var c = counter(), d = counter(); // Create two counters

c.count() // => 0

d.count() // => 0: they count independently

c.reset() // reset() and count() methods share state

c.count() // => 0: because we reset c

d.count() // => 1: d was not reset

The counter() function returns a “counter” object. This object has two methods:

count() returns the next integer, and reset() resets the internal state. The first thing to

understand is that the two methods share access to the private variable n. The second

thing to understand is that each invocation of counter() creates a new scope chain and

a new private variable. So if you call counter() twice, you get two counter objects with

different private variables. Calling count() or reset() on one counter object has no

effect on the other.

function counter(n) { // Function argument n is the private variable

return {

// Property getter method returns and increments private counter var.

get count() { return n++; },

// Property setter doesn't allow the value of n to decrease

set count(m) {

if (m >= n) n = m;

else throw Error("count can only be set to a larger value");

}

};

}

var c = counter(1000);

c.count // => 1000

c.count // => 1001

c.count = 2000

c.count // => 2000

c.count = 2000 // => Error!

// This function adds property accessor methods for a property with

// the specified name to the object o. The methods are named get<name>

// and set<name>. If a predicate function is supplied, the setter

// method uses it to test its argument for validity before storing it.

// If the predicate returns false, the setter method throws an exception.

//

// The unusual thing about this function is that the property value

// that is manipulated by the getter and setter methods is not stored in

// the object o. Instead, the value is stored only in a local variable

// in this function. The getter and setter methods are also defined

// locally to this function and therefore have access to this local variable.

// This means that the value is private to the two accessor methods, and it

// cannot be set or modified except through the setter method.

function addPrivateProperty(o, name, predicate) {

var value; // This is the property value

// The getter method simply returns the value.

o["get" + name] = function() { return value; };

// The setter method stores the value or throws an exception if

// the predicate rejects the value.

o["set" + name] = function(v) {

if (predicate && !predicate(v))

throw Error("set" + name + ": invalid value " + v);

else

value = v;

};

}

// The following code demonstrates the addPrivateProperty() method.

var o = {}; // Here is an empty object

// Add property accessor methods getName and setName()

// Ensure that only string values are allowed

addPrivateProperty(o, "Name", function(x) { return typeof x == "string"; });

o.setName("Frank"); // Set the property value

console.log(o.getName()); // Get the property value

o.setName(0); // Try to set a value of the wrong type

// This function returns a function that always returns v

function constfunc(v) { return function() { return v; }; }

// Create an array of constant functions:

var funcs = [];

for(var i = 0; i < 10; i++) funcs[i] = constfunc(i);

// The function at array element 5 returns the value 5.

funcs[5]() // => 5

When working with code like this that creates multiple closures using a loop, it is a

common error to try to move the loop within the function that defines the closures.

Think about the following code, for example:

// Return an array of functions that return the values 0-9

function constfuncs() {

var funcs = [];

for(var i = 0; i < 10; i++)

funcs[i] = function() { return i; };

return funcs;

}

var funcs = constfuncs();

funcs[5]() // What does this return?

The code above creates 10 closures, and stores them in an array. The closures are all

defined within the same invocation of the function, so they share access to the variable

i. When constfuncs() returns, the value of the variable i is 10, and all 10 closures share

this value. Therefore, all the functions in the returned array of functions return the same

value, which is not what we wanted at all. It is important to remember that the scope

chain associated with a closure is “live.” Nested functions do not make private copies

of the scope or make static snapshots of the variable bindings.