JavaScript Language Specification

JavaScript 1.1

10/3/96

Contents

11
11

15
15

Chapter

1

1 Introduction

JavaScript is a compact, platform-indpendent, object-based scripting language.

1.1 Versions and Implementations

JavaScript version 1.0 was implemented in Netscape Navigator 2.0 and Netscape LiveWire 1.0. This specification describes JavaScript version 1.1, which was implemented in Netscape Navigator 3.0. This specification describes the language, and not the specific features of any particular implementation.

Although JavaScript was originally implemented as a scripting language supplement to HTML in Netscape Navigator and LiveWire, this specification does not prescribe any particular application; the language itself is flexible and apt for many applications.

1.2 Grammar Notation

Terminal symbols are shown in `fixed width` font in the productions of the lexical and syntactic grammars, and throughout this specification whenever the text is directly referring to such a terminal symbol. These are to appear in a program exactly as written.

Nonterminal symbols are shown in *`italic fixed width font`*. The definition of a nonterminal is introduced by the name of the nonterminal followed by a colon. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. For example, the syntactic definition:

```
IfThenStatement:
     if ( Expression ) Statement
```

states that the nonterminal *IfThenStatement* represents the token `if`, followed by a left parenthesis token, followed by an Expression, followed by a right parenthesis token, followed by a Statement. As another example, the syntactic definition:

```
ArgumentList:
     Argument
     ArgumentList , Argument
```

states that an *ArgumentList* may represent either a single *Argument* or an *ArgumentList*, followed by a comma, followed by an *Argument*. This definition of *ArgumentList* is recursive, that is to say, it is defined in terms of itself. The result is that an *ArgumentList* may contain any positive number of arguments. Such recursive definitions of nonterminals are common.

The subscripted suffix $_{\text{opt}}$, which may appear after a terminal or nonterminal, indicates an optional symbol. The alternative containing the optional symbol actually specifies two right-hand sides, one that omits the optional element and one that includes it. So, for example:

```
ReturnStatement:
     return Expression_opt
```

is an abbreviation for:

```
ReturnStatement:
     return
     return Expression
```

When the words "one of" follow the colon in a grammar definition, they signify that each of the terminal symbols or tokens on the following line or lines is an alternative definition. For example:

```
OctalDigit: one of
     0 1 2 3 4 5 6 7
```

which is a convenient abbreviation for:

```
ZeroToThree:
     0
     1
     2
     3
     4
     5
     6
     7
```

The right-hand side of a lexical production may indicate that certain expansions are not permitted by using the phrase "but not" and then naming the excluded expansions, as in the productions for *Identifier*:

```
Identifier:
     IdentifierName, but not a Keyword or BooleanLiteral or NullLiteral
```

Finally, a few nonterminal symbols are described by a descriptive phrase where it would be impractical to list all the alternatives, for example:

```
RawInputCharacter:
     any ASCII character
```

1.3 Example Programs

The example programs given in the text are ready to be executed by a JavaScript system. Since this specification does not describe any specific mechanism for JavaScript to display output, examples suppose a simple `println` function that displays values to the user. In Netscape Navigator, this function would be defined as follows:

```
function println(x) {
     document.write(x, "<BR>")
}
```

This function is intended for illustrative and pedagogical purposes only, and is not part of the language specification.

1.4 References

Bobrow, Daniel G., Linda G. Demichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. *Common Lisp Object System Specification, X3J13 Document 88-002R,* June 1988; appears as Chapter 28 of Steele, Guy. *Common Lisp: The Language, 2nd ed.* Digital Press, 1990, ISBN 1-55558-041-6, 770-864.

IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std. 754-1985. Available from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112-5704 USA; 800-854-7179.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language, 2nd ed*. Prentice Hall, Englewood Cliffs, New Jersey, 1988, ISBN 0-13-110362-8.

Gosling, James, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley Publishing Company, 1996

Cardelli, et. al. *Modula-3 Report (revised)*. Digital Equipment Corporation, 1989, Palo Alto, California

Agesen, et. al. *The Self 3.0 Programmer's Reference Manual.* Sun Microsystems, 1993, Mountain View, California.

Stroustrup, Bjarne. *The C++ Progamming Language, 2nd ed.* Addison-Wesley, Reading, Massachusetts, 1991, reprinted with corrections January 1994, ISBN 0-201-53992-6.

Chapter

2

2 Lexical Structure

This chapter describes the lexical structure of JavaScript by specifying the language’s lexical grammar.

2.1 ASCII

JavaScript programs are written using *ASCII*, the American Standard Code for Information Interchange (defined by ANSI standard X3.4).

2.2 Lexical Translations

This chapter describes the translation of a raw ASCII character stream into a sequence of JavaScript tokens, using the following two lexical translations, which are applied in turn:

1. A translation of the ASCII character stream into a stream of input characters and line terminators.

2. A translation of the stream of input characters and line terminators into a sequence of JavaScript input elements which, after white space and comments are discarded, comprise the tokens that are the terminal symbols of the syntactic grammar for JavaScript.

In these lexical translations JavaScript chooses the longest possible translation at each step, even if the result does not ultimately make a correct JavaScript program, while another lexical translation would.

2.3 Line Terminators

JavaScript divides the sequence of input characters into lines by recognizing *line terminators*. This definition of lines determines the line numbers produced by a JavaScript compiler or other system component. It also specifies the termination of a single-line comment.

Lines are terminated by the ASCII characters CR, or LF, or CR LF. A CR immediately followed by LF is counted as one line terminator, not two.

```
RawInputCharacter:
     LineTerminator
     InputCharacter

LineTerminator:
     the ASCII LF character, also known as "newline"
     the ASCII CR character, also known as "return"
     the ASCII CR character followed by the ASCII LF character

InputCharacter:
     Any ASCII character, but not CR and not LF
```

The result of this step is a sequence of line terminators and input characters, which are the terminal symbols for the second step in the tokenization process.

2.4 Input Elements and Tokens

The input characters and line terminators that result from input line recognition are reduced to a sequence of *input elements*. The input elements that are not white space or comments are JavaScript *tokens*.

This process is specified by the following grammar:

```
InputElements:
     InputElement$_{opt}$
     InputElements InputElement

InputElement:
     WhiteSpace
     Comment
     Token
```

```
WhiteSpace:
     the ASCII SP character, also known as "space"
     the ASCII HT character, also known as "horizontal tab"
     the ASCII FF character, also known as "form feed"
     LineTerminator
```

```
Token:
     Keyword
     Identifier
     Literal
     Separator
     Operator
```

White space and comments can serve to separate tokens that, if adjacent, might be tokenized in another manner. For example, the characters `-` and `=` in the input can form the operator token `-=` only if there is no intervening white space or comment.

2.4.1 White Space

White space is defined as the ASCII space, horizontal tab, and form feed characters, as well as line terminators.

2.4.2 Semicolons

The ASCII Semicolon character (;) separates multiple statements on a single line. Unlike Java, a semicolon is not required to terminate a statement. This is equivalent to compile-time error-correction in which JavaScript inserts semicolons as neccessary.

Complete lexcical specification of semicolons TBD.

2.4.3 Comments

JavaScript has two kinds of *comments*:

A traditional comment: all the text from `/*` to `*/` is ignored (as in C):

```
/* text */
```

A single-line comment: all the text from `//` to the end of the line is ignored (as in C++):

```
// text
```

These comments are formally specified by the following lexical grammar:

```
Comment:
     TraditionalComment
     SingleLineComment

TraditionalComment:
     /* CommentTextopt */

CommentText:
     CommentCharacter
     CommentText CommentCharacter

CommentCharacter:
     NotStarSlash
     / NotStar
     * NotSlash
     LineTerminator

NotStar
     InputCharacter, but not *

NotSlash
     InputCharacter, but not /

NotStarSlash
     InputCharacter, but not * and not /

SingleLineComment:
     // CharactersInLineopt LineTerminator

CharactersInLine:
     InputCharacter
     CharactersInLine InputCharacter
```

The grammar implies all of the following properties:

- Multi-line comments cannot be nested
- /* and */ have no special meaning in // comments.
- // has no special meaning in either single-line or multi-line comments

As a result, these are legal comments:

```
/* this comment // ends here: */
// This // just /* fine */ as far as JavaScript // is concerned
```

But this causes a compile-time warning:

```
/* this comment /* causes a compile-time warning
```

2.5 Keywords

The following sequences of ASCII letters are reserved for use as *keywords*, and are not legal identifiers:

```
Keyword: one of

       abstract      else          instanceof    super
       boolean       extends       int           switch
       break         false         interface     synchronized
       byte          final         long          this
       case          finally       native        throw
       catch         float         new           throws
       char          for           null          transient
       class         function      package       true
       const         goto          private       try
       continue      if            protected     var
       default       implements    public        void
       do            import        return        while
       double        in            short         with
                                   static
```

The above list includes all keywords used and reserved for future use. The following table lists keywords currently used in JavaScript:

```
break         new
continue      null
delete        return
else          this
false         true
for           var
function      void
if            while
in            with
```

While `true` and `false` might appear to be keywords, they are technically Boolean literals; while `null` might appear to be a keyword, it is technically an object reference.

2.6 Identifiers

An *identifier* is an unlimited length sequence of ASCII *letters* and *digits*, the first of which must be a letter. The letters include uppercase and lowercase ASCII letters (a-z and A-Z) and the ASCII underscore (_) and dollar sign ($). The digits include the ASCII digits 0-9.

```
Identifier:
     IdentifierName, but not a Keyword or BooleanLiteral or NullLiteral

IdentifierName:
     JavaScriptLetter GraphicCharacter

JavaScriptLetter:
     any uppercase or lowercase ASCII letter (a-z, A-Z)
     any digit (0-9)
     _
     $

GraphicCharacter:
     any symbol in the set: ~‘!@#%^&*()-+={[}]|\:”’<,>.?/
```

Examples of legal identifiers are

```
     Number_hits
     temp99
     _name
     $6million
```

2.7 Literals

A *literal* is the source code representation of a value of a primitive type:

```
Literal:
     IntegerLiteral
     FloatingPointLiteral
     BooleanLiteral
     StringLiteral
     NullLiteral
```

2.7.1 Integer Literals

Integer literals may be expressed in decimal (base 10), hexadecimal (base 16), or octal (base 8):

```
IntegerLiteral:
     DecimalLiteral
     HexLiteral
     OctalLiteral
```

A decimal literal consists of a digit from `1` to `9`, optionally followed by one or more digits from `0` to `9`, and represents a positive integer:

```
DecimalLiteral:
     0
     NonZeroDigit Digits_opt

Digits:
     Digit
     Digits Digit

Digit:
     0
     NonZeroDigit

NonZeroDigit: one of
     1 2 3 4 5 6 7 8 9
```

A hexadecimal literal consists of a leading `0x` or `0X` followed by one or more hexadecimal digits and can represent a positive, zero, or negative integer. Hexadecimal digits with values 10 through 15 are represented by the letters `a` through `f` or `A` through `F`, respectively; each letter used as a hexadecimal digit may be uppercase or lowercase.

```
HexLiteral:
     0x HexDigit
     0X HexDigit
     HexLiteral HexDigit

HexDigit: one of
     0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

An octal literal consists of a digit `0` followed by one or more of the digits `0` through `7` and can represent a positive, zero, or negative integer.

```
OctalLiteral:
     0 OctalDigit
     OctalLiteral OctalDigit

OctalDigit: one of
     0 1 2 3 4 5 6 7
```

The largest positive hexadecimal and octal literals of type `int` are `0x7fffffff` and `017777777777`, respectively, which equal `2147483647`. The most negative hexadecimal and octal literals of type `int` are `0x80000000` and `020000000000` respectively, each of which represents the decimal value `-2147483648`. The hexadecimal and octal literals `0xffffffff` and `037777777777`, respectively, represent the decimal value `-1`.

Examples of integer literals:

```
0
2
0372
0xDeadBeef
1996
0x00FF00FF
```

2.7.2 Floating-Point Literals

A floating-point literal has the following parts: a whole-number part, a decimal point, a fractional part, an exponent, and a type suffix. The exponent, if present, is indicated by a letter `e` or `E` followed by an optionally signed integer.

At least one digit, in either the whole number or the fraction part, and either a decimal point, an exponent, or a float type suffix are required. All other parts are optional.

FloatingPointLiteral:
    Digits . Digits$_{opt}$ *ExponentPart*$_{opt}$
    . Digits ExponentPart$_{opt}$
    Digits ExponentPart
    Digits ExponentPart$_{opt}$

ExponentPart:
    ExponentIndicator SignedInteger

ExponentIndicator: one of
    e E

SignedInteger:
    Sign$_{opt}$ *Digits*

Sign: one of
    + -

The largest positive finite number is 1.79769313486231570e+308. Numbers greater than this are represented by the literals `+Infinity` (for positive numbers) and `-Infinty` (for negative numbers).

The constant NaN represents a value that is Not-a-Number.

Examples of floating-point literals are:

```
1e1
2.
.3
0.0
3.14
1e-9
```

2.7.3 Boolean Literals

The `boolean` type has two values: `true` and `false`.

BooleanLiteral:
    `true`
    `false`

2.7.4 String Literals

A string literal is zero or more characters, enclosed in single (’) or double (")quotes.

StringLiteral:
    " *StringCharactersDQ*$_{opt}$ "
    ’ *StringCharactersSQ*$_{opt}$ ’

StringCharactersDQ:
    StringCharacterDQ
    StringCharactersDQ StringCharacterDQ

StringCharactersSQ:
    StringCharacterSQ
    StringCharactersSQ StringCharacterSQ

StringCharacterDQ:
    InputCharacter, but not " or \
    EscapeSequence

StringCharacterSQ
    InputCharacter, but not ’ or \
    EscapeSequence

The escape sequences are described in section 2.7.5 Escape Sequences for String Literals.

It is a compile-time error for a line terminator to appear after the opening " and before the closing matching ". A long string literal can always be broken up into shorter pieces and written as a (possibly parenthesized) expression using the string concatenation operator +.

Examples of string literals:

```
""                               // The empty string
"\""                             // A string containing " alone
‘This is a string’               // A string containing 16 characters

"This is a " +                   // Actually a string-valued expression
        "two-line string"   // containing two string literals
```

2.7.5 Escape Sequences for String Literals

The string *escape sequences* allow for the representation of some nongraphic characters as well as the single quote, double quote, and backslash characters in string literals.

```
EscapeSequence:
     \ b (backspace BS)
     \ t (horizontal tab HT )
     \ n (linefeed LF )
     \ f (form feed FF )
     \ r (carriage return CR )
     \ " (double quote " )
     \ ' (single quote ' )
     \ \ (backslash \ )

OctalEscape:
     \ OctalDigit
     \ OctalDigit OctalDigit
     \ ZeroToThree OctalDigit OctalDigit

OctalDigit: one of
     0 1 2 3 4 5 6 7

HexEscape:

     \ xHexDigit HexDigit
     \ XHexDigit HexDigit

HexDigit: one of
     0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

2.7.6 The Null Literal

The null object reference has one value, denoted by the literal `null`.

```
NullLiteral:
    null
```

2.8 Separators

The following characters are used in JavaScript as *separators* (punctuators):

```
Separator: one of

     ( ) { } [ ] , .
```

2.9 Operators

The following tokens are used in JavaScript as *operators*:

```
Operator: one of
= > < ! ~ ? :
== <= >= != && || ++ --
+ - * / & | ^ % << >> >>>
+= -= *= /= &= |= ^= %= <<= >>= >>>=
```

Chapter

3

3 Types, Values, and Variables

JavaScript is a *dynamically-typed* language, which means that the datatypes of variables are not declared, and the datatypes of variables and expressions are determined at run time.

3.1 Types

There are two kinds of types in JavaScript: primitive types and reference types. There are, correspondingly, two kinds of data values that can be stored in variables, passed as arguments, returned by methods, and operated on: primitive values and reference values.

```
Type:

     PrimitiveType
     ReferenceType
```

JavaScript's primitive data types are `boolean` and `number`; its reference types are `string`, `object` (including the `null` object), and `function`. Strings have primitive equality and relational operator semantics.

The `boolean` type has the truth values `true` and `false`. A `number` can be either an integer or floating-point; JavaScript does not explicitly distinguish between them. Integers are 32-bit signed two's-complement integers and floating-point numbers are 64-bit IEEE 754 floating-point numbers.

An `object` in JavaScript is a container that associates names and indexes with data of arbitrary type. These associations are called properties. Properties with function values are called the object’s *methods*.

3.1.1 Type Names

The **typeof** operator returns a string indicating the type of its operand. The string returned is one of

- “undefined”
- “object”
- “function”
- “number”
- “boolean”
- “string”

For more information on **typeof**, see 4.17.3 The typeof operator.

3.1.2 Type Conversion

As summarized in Table 3.1, JavaScript performs automatic type conversion at run-time. Conversions are performed as neccessary.

More information TBD.

			To type		
From type	**function**	**object**	**number**	**boolean**	**string**
undefined	error	null	error	false	“undefined”
function	N/C	Function object	error	error	decompile
object					
(not null)	Function object	N/C	valueof	valueof/true	toString/valueOf/default*
(null)	error		0	false	“null”
number					
(zero)	error	null	N/C	false	“0”
(nonzero)	error	Number		true	default*
(NaN)	error	Number		false	“NaN”
(+Infinity)	error	Number		true	“+Infinity”
(-Infinity)	error	Number		true	“-Infinity”
Boolean					
(false)	error	boolean	0	N/C	“false”
(true)	error	boolean	1		“true”
string					
(empty)	error	String	error	false	N/C
(non-empty)	error	String	number (if string is numeric literal) else error	true	

Key:

N/C - No conversion neccessary.
decompile - a string containing the function’s canonical source
toString - the result of the toString method
valueOf - The valueof function is tried, and if it returns a value, that value is used.
*default:
object - [object *name*] where *name* is constructor function name
number - Numeric value parsed from string, if possible; see Number.toString()

Explanation of key TBD.

3.1.3 The toString method

Every object has a toString method that is used to convert the object to a string value: boolean, and number types are converted to their string equivalents. Functions are decompiled, that is, a string is returned that is the function definition. Objects return a string that is “[object Object]”.

Examples

For example,

```
function f() {
     return 42
}

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

objnull = null

o = new Car("Ford", "Mustang", 1969)
posInfinity = 10*1e308
d = new Date()
n0 = 0
n1 = 123

println(true.toString())
println(false.toString())
println(f.toString())
println(objnull)
println(d.toString)
println(Math.toString)
println(o.toString())
println(n0.toString())
println(n1.toString())
println(posInfinity.toString())
```

This script returns the following:

```
true
false
function f() {
     return 42
}
null
function toString() { [native code] }
function toString() { [native code] }
```

```
[object Object]
0
123
Infinity
```

3.1.4 The valueOf Method

Every object has a valueOf method that returns the primitive value associated with the object, if any. If there is no primitive value, valueOf returns the object itself. For Number and Boolean objects, valueOf returns the primitive numeric or Boolean value of the object, respectively.

3.2 Primitive Types and Values

A primitive type is predefined by the JavaScript language:

```
PrimitiveType:
     number
     boolean
     undefined
```

Datatypes are not declared, and a variable can change type as different values are assigned to it. A number can be either an integer or a floating-point number.

3.2.1 Numeric Types and Values

JavaScript numbers are signed 64-bit IEEE 754 floating-point values, as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985 (IEEE, New York).

The IEEE 754 standard includes not only positive and negative sign-magnitude numbers, but also positive and negative *infinities*, and a special *Not-a-Number* (hereafter abbreviated NaN). The NaN value is used to represent the result of certain operations such as dividing zero by zero.

The largest positive finite number is 1.79769313486231570e+308. The smallest positive finite nonzero number is 4.94065645841246544e-324.

Except for NaN, numeric values are *ordered*; arranged from smallest to largest, they are negative infinity, negative finite values, negative zero, positive zero, positive finite values, and positive infinity.

NaN is *unordered*, so the numerical comparison operators <, <=, >, and >= return `false` if either or both of their operands are NaN . The numerical equality operator == returns `false` if either operand is NaN, and the inequality operator `!=` returns `true` if either operand is NaN . In particular, `x==x` is `false` if and only if `x` is NaN, and `(x<y)==!(x>=y)` will be `false` if `x` or `y` is NaN.JavaScript provides a number of operators that act on numeric values. These operators treat their operands as 64-bit floating-point values.

3.2.2 Numeric Operations

JavaScript provides a number of operators that act on numeric values:

- The comparison operators, which result in a value of type `boolean`:
  - The numerical comparison operators <, <=, >, and >=
  - The numerical equality operators == and `!=`
- The unary minus operator `-`
- The multiplicative operators `*`, `/`, and `%`
- The additive operators + and `-`
- The modulus operator %
- The increment operator ++, both prefix and postfix
- The decrement operator `--`, both prefix and postfix
- The conditional operator `? :`

Other useful methods and constants are defined for the Math object.

Numeric operators behave as specified by IEEE 754. In particular, JavaScript requires support of IEEE 754 *denormalized* floating-point numbers and *gradual underflow*, which make it easier to prove desirable properties of particular numerical algorithms.

JavaScript requires that floating-point arithmetic behave as if every floating-point operator rounded its floating-point result to the result precision. *Inexact* results must be rounded to the representable value nearest to the infinitely precise result; if the two nearest representable values are equally near, the one with its least significant bit zero is chosen. This is the IEEE 754 standard's default rounding mode known as *round to nearest.*

An operation that overflows produces a signed infinity, an operation that underflows produces a signed zero, and an operation that has no mathematically definite result produces NaN. All numeric operations with NaN as an operand produce NaN as a result. Since NaN is unordered, a numeric comparison operation involving one or two NaNs returns `false` and any `!=` comparison involving NaN returns `true`, including `x!=x` when `x` is NaN.

The following example illustrates:

```
// an example of overflow:
d = 1e308
println("overflow produces infinity: ")
println(d + "*10==" + d*10)
println("")

// an example of gradual underflow:
d = 1e-305 * Math.PI
println("gradual underflow: ")
println(d)
for (i = 0; i < 4; i++)
     println(d /= 100000)
println("")

// an example of NaN:
d = 0.0/0.0
println("0.0/0.0 is Not-a-Number: ", d)
println("")

// an example of inexact results and rounding:
println("inexact results with floating point arithmetic:")
for (i = 0; i < 100; i++) {
     z = 1.0/i
     if (z*i != 1.0)
          println(i)
}
```

This example produces the following output:

```
overflow produces infinity:
1e308*10==Infinity

gradual underflow:
3.141592653589793e-305
```

```
3.1415926535898e-310
3.141592653e-315
3.142e-320
0

0.0/0.0 is Not-a-Number: NaN

inexact results with floating point arithmetic:
49
98
```

This example demonstrates, among other things, that gradual underflow can result in a gradual loss of precision. Note that when `i` is zero, `z` is `NaN`, and `z*i` is NaN.

3.2.2.1 Bitwise integer operations

The bitwise operators treat their operands as signed 32-bit integer values:

- The signed and unsigned shift operators <<, >>, and >>>
- The bitwise complement operator ~
- The integer bitwise operators &, `|`, and `^`

JavaScript uses *round toward zero* when converting a floating-point value to an integer, which acts, in this case, as though the number were truncated, discarding the mantissa bits. Round toward zero chooses the value closest to and no greater in magnitude than the infinitely precise result.

3.2.3 Boolean Types and Values

The `boolean` type represents a logical quantity with two possible values, the literals `true` and `false`.

3.2.4 Boolean Operations

JavaScript's Boolean operators treat their operands as Boolean values and return a Boolean value:

- The logical operator `!`

- The logical-or and logical-and operators && and `||`
- The conditional operator `? :`

Boolean expressions control the control flow in:

- The `if` statement
- The `while` statement
- The `for` statement

When they do so, they determine which subexpression is chosen to be evaluated in the conditional `? :` operator .

Only Boolean expressions can be used in the control flow statements and as the first operand of the conditional operator `? :`. A number is converted to a Boolean value following the C language convention that zero is `false` and any nonzero value is `true`, An object is converted to a Boolean, following the C language convention that the `null` object is false and any object other than `null` is `true`.

3.2.5 The Undefined Type

Any variable that has not been assigned a value is automatically of type undefined.

3.3 Reference Types and Values

JavaScript’s reference types are objects, strings, and functions.

```
ReferenceType
     Object
     String
     Function
```

3.3.1 Object Types and Operations

An object is a dynamically created reference with members that can be primitive values, objects, or functions. The reference values are pointers to these objects, and a special `null` reference, which refers to no object.

An object is created with the **new** operator operating on a constructor, either one of the built-in constructors (Array, Date, Math, and String) or a user-defined constructor function (see 5.3 Constructor Functions).

3.3.1.1 The Null Object

The null object is a special object that references no object. It is named by the `null` literal.

3.3.1.2 The Delete Operator

The **delete** operator removes a property definition, frees the memory associated with it, and returns undefined.

3.3.1.3 The Void Operator

The operator **void** returns undefined after evaluating its operand.

3.3.2 String Types and Operations

A string is a special object created in one of two ways: either by using a string literal or by the use of the **new** operator with the String constructor.

Every string has a `length` property that is an integer equal to the number of characters in the string. Strings also have a number of built-in methods, as described in 7.7 String Object.

The following operators are defined for strings:

- The concatentation (+) operator that concatentates two strings together. If given a `String` operand and a floating-point operand, it will convert the floating-point operand to a string representing its value in decimal form, and then produce a new string that is the concatenation of the two strings.

- Relational and equality operators (==, >, <, >=, <=) that compare the lexicographical precedence of their operands and return a logical value.

Note the reference-type behavior for assignment operations, but not for equality and relational operations.

3.3.3 Function Types

A function is created by a *function definition*. The syntax for a function definition is given in 6.4.10 Function Definition Statement.

It is also possible to create a *function object* with the **new** operator as follows:

```
functionObject:
     identifierName = new Function("block")
     identifierName = new Function(parameterList, "block")

parameterList:
     "identifierName"
     "identifierName", parameterList
```

where *block* is the set of statements that defines the body of the function.

Examples

Here is a standard declaration of a simple factorial function:

```
function factorial(n) {
     if ((n <= 1))
          return 1
     else
          return (n * factorial(n-1) )
}
```

Here is the same function declared as a function object:

```
myfactorial = new Function("n", "if ((n <= 1)) return 1; else return n *
factorial(n-1);" )
```

3.4 Variables

A variable is a storage location for a value and has an associated type, determined at run-time. A variable's value is changed by an assignment, by a prefix or postfix `++` (increment) or `--` (decrement) operator, or by the delete operator.

3.4.1 Kinds of Variables

JavaScript has two kinds of variables: standard variables and function parameters.

Standard variables are declared by use or by variable declaration statements.

Function parameters name argument values passed to a function. For every parameter declared in a function declaration, a new parameter variable is created each time that function is invoked . The new variable is initialized with the corresponding argument value from the function invocation. If the function invocation does not specify a value for the parameter, then it has a value of `undefined`. The function parameter effectively ceases to exist when the execution of the body of the function is complete.

3.4.2 Initial Values of Variables

Every variable in a JavaScript program has a value:

- If the variable is assigned a value in its declaration or its initial use, then it has that value, otherwise its value is `undefined`.
- Each function parameter is initialized to the corresponding argument value provided in the invocation of the function, otherwise its value is `undefined`.

A local variable must be explicitly given a value (other than `undefined`) before it is used, by either initialization or assignment, or a run-time error results.

3.5 Names

A *name* is used to refer to an entity declared in a JavaScript program. A declared entity is a variable or a function. There are two forms of names: simple names and qualified names. A *simple name* is a single identifier. A *qualified name* consists of a name, a "**.**" token, and an identifier; it is used when an identifier is associated with an object.

In determining the meaning of a name, JavaScript takes into account the context in which the name appears. It distinguishes among contexts where a name must denote (refer to) a type, a variable or value in an expression, or a function.

Not all identifiers in JavaScript programs are a part of a name. Identifiers are also used in the following situations:

- In declarations, where an identifier occurs to specify the name by which the declared entity will be known
- In property expressions, where an identifier occurs after a "**.**" token to indicate a member of an object that is the value of an expression
- In method invocation expressions, where an identifier occurs after a "**.**" token and before a "**(**" token to indicate a method to be invoked for an object that is the value of an expression

3.5.1 Declarations and Scoping

JavaScript variables may be declared by assignment or by use of the **var** statement. For example, both of the following statements declare the variable x:

```
x = 42
var x = 42
```

A variable declared outside a function is a *global variable*, and is accessible everywhere in the global scope. A variable declared by assignment inside a function is also a global variable. A variable declared with **var** inside a function is a *local variable*, and is accessible only within that function.

3.5.2 Hiding Names

When there is a global variable with the same name as a local variable, the local variable is said to *hide* the global variable within the function. In this case, there are two variables, a global variable and a local variable, with the same names. Within the function, the local variable is used; everywhere else in the application, the global variable takes precedence.

For example, in the function foo, x has a value of 17, but outside the function, it has a value of 42.

```
x = 42

function foo() {
     var x = 17
     println(x)
}

foo()
println(x)
```

The result of these statements is:

```
17
42
```

Chapter

4

4 Expressions

This chapter specifies the meaning of JavaScript expressions and the rules for their evaluation.

4.1 Evaluation, Denotation and Result

When a JavaScript expression is *evaluated* (executed), the *result* denotes one of three things:

- a variable (in C, this would be called an *lvalue*)
- a value
- undefined (the expression is said to be `void`)

Evaluation of an expression can also produce side effects, because expressions may contain embedded assignments, increment or decrement operators, and function invocations.

An expression returns void if it gets its value from a function invocation that does not return a value or from use of the void operator.

If an expression denotes a variable, and a value is required for use in further evaluation, then the value of that variable is used. In this context, when the expression denotes a variable or a value, we may speak simply of the *value* of the expression.

4.2 Evaluation Order

In JavaScript, the operands to operators are evaluated from left to right.

The left-hand operand of a binary operator is fully evaluated before any part of the right-hand operand is evaluated. For example, if the left-hand operand contains an assignment to a variable and the right-hand operand contains a reference to that same variable, then the value produced by the reference will reflect the fact that the assignment occurred first.

Thus:

```
if ((obj = foo.bar) && obj.flag)
     println(“True”)
```

Need clarification of this example.

Every operand of an operator (except for the conditional operators &&, `||`, and `? :`) is fully evaluated before any part of the operation itself is performed.

In a function or constructor invocation, one or more argument expressions may appear within the parentheses, separated by commas. Each argument expression is fully evaluated before any part of any argument expression to its right is evaluated.

4.3 Primary Expressions

Primary expressions include most of the simplest kinds of expressions, from which all others are constructed: literals, function invocations, and array accesses.

```
PrimaryExpr:
     Literal
     this
     FunctionInvocation
     ArrayAccess
```

4.3.1 Literals

A literal denotes a fixed, unchanging value.

The following production from Chapter 2 is repeated here for convenience:

```
Literal:
     IntegerLiteral
     FloatingPointLiteral
     BooleanLiteral
     StringLiteral
     NullLiteral
```

4.3.2 this

The keyword `this` denotes a reference to the invoking object, or to the object being constructed in a constructor function. The invoking object is defined as the object name to the left of the period "." or left bracket "[" in a method invocation, otherwise it is the parent object of the function.

4.4 Function Invocation Expressions

A function invocation expression is used to invoke functions, including methods of objects and constructors:

FunctionInvocation:
    FunctionName (ArgumentList$_{opt}$ *)*
    ObjectName . FunctionName(ArgumentList$_{opt}$ *)*

The definition of *ArgumentList* is repeated here for convenience:

```
ArgumentList:
     Expression
     ArgumentList , Expression
```

The following sections describe the processing of a function invocation.

4.4.1 Runtime Evaluation

At run time, function invocation requires the following steps:

- A *target reference* may be computed.
- The argument expressions are evaluated.
- The actual code for the function is executed.

4.4.1.1 Compute Target Reference

Determine the function to be executed, depending on the form of syntax used. If the form was a top-level invocation, i.e. *`FunctionName`*, then the compiler searches for the definition of the specified function.

If the form was that of a method invocation, i.e. *`ObjectName.FunctionName`*, then the object definitions are first searched for the method definition (containing a reference to a function), and then the specified function definition is searched for.

4.4.1.2 Evaluate Arguments

The argument expressions are evaluated in order, from left to right.

4.4.1.3 Locate Function to Invoke

The body of the target function identified in the first step is executed, with the values of the arguments determined in the second step.

The arguments in the function invocation expression are paired with the corresponding formal arguments in the function definition. If there are fewer arguments than in the function definition, then the remaining formal arguments are undefined during the current invocation. If there are more arguments in the invocation expression than in the definition, these arguments are assigned to elements of the function's arguments array.

4.5 Postfix Expressions

Postfix expressions include uses of the postfix `++` and `--` operators. Also, names are not considered to be primary expressions, but are handled separately in the grammar to avoid certain ambiguities. They become interchangeable only here, at the level of precedence of postfix expressions.

```
PostfixExpression:
     PrimaryExpr
     PostIncrementExpression
     PostDecrementExpression
```

4.5.1 Postfix Increment Operator ++

```
PostIncrementExpression:
     PostfixExpression ++
```

A postfix expression followed by a `++` operator is a postfix increment expression. The operand must be convertible into a number, and thus the result of the operation must be a variable of type number or a run-time error occurs. The result of the postfix increment expression is not a variable, but a value.

At run time, the value `1` is added to the value of the variable and the sum is stored back into the variable. The value of the postfix increment expression is the value of the variable *before* the new value is stored.

4.5.2 Postfix Decrement Operator --

```
PostDecrementExpression:
     PostfixExpression --
```

A postfix expression followed by a `--` operator is a postfix decrement expression. The operand must be convertible into a number, and thus the result of the operation must be a variable of type number or a run-time error occurs. The result of the postfix decrement expression is not a variable, but a value.

At run time, the value `1` is subtracted from the value of the variable and the difference is stored back into the variable. The value of the postfix decrement expression is the value of the variable *before* the new value is stored.

4.6 Unary Operators

The unary operators include `+`, `-`, `++`, `--`, ~, and `!`. Expressions with unary operators group right-to-left, so that -~x means the same as -(~x).

```
UnaryExpression:
     PreIncrementExpression
     PreDecrementExpression
     - UnaryExpression
     UnaryExpressionNotPlusMinus

PreIncrementExpression:
     ++ UnaryExpression

PreDecrementExpression:
     -- UnaryExpression

UnaryExpressionNotPlusMinus:
     PostfixExpression
     ~ UnaryExpression
     ! UnaryExpression
```

4.6.1 Prefix Increment Operator ++

A unary expression preceded by a `++` operator is a prefix increment expression. The result of the unary expression must be a variable type number or a run-time error occurs. The result of the prefix increment expression is not a variable, but a value.

At run time, the value `1` is added to the value of the variable and the sum is stored back into the variable. The value of the prefix increment expression is the value of the variable *after* the new value is stored.

4.6.2 Prefix Decrement Operator --

A unary expression preceded by a `--` operator is a prefix decrement expression. The result of the unary expression must be a variable of type `number` or a run-time error occurs. The result of the prefix decrement expression is not a variable, but a value.

At run time, the value `1` is subtracted from the value of the variable and the difference is stored back into the variable. The value of the prefix decrement expression is the value of the variable *after* the new value is stored.

4.6.3 Unary Minus Operator -

The operand expression of the unary - operator must be convertible to a number or a run-time error occurs. At run-time, the value of the unary plus expression is the arithmetic negation of the converted value of the operand.

For integer values, negation is the same as subtraction from zero. JavaScript uses two's-complement representation for integers, and the range of two's-complement values is not symmetric, so negation of the maximum negative int or long results in that same maximum negative number. Overflow occurs in this case. For all integer values x, -x equals (~x)+1.

Special cases are:

- If the operand is NaN, the result is NaN (recall that NaN has no sign).
- If the operand is an infinity, the result is the infinity of opposite sign.

4.6.4 Bitwise Complement Operator ~

The type of the operand expression of the unary ~ operator must be a convertible to a number or a run-time error occurs. At run time, the value of the unary bitwise complement expression is the bitwise complement of the promoted value of the operand; note that, in all cases, ~x equals (-x)-1.

4.6.5 Logical Complement Operator !

The type of the operand expression of the unary ! operator must be convertible to a boolean value or a run-time error occurs. The type of the unary logical complement expression is boolean.

At run time, the value of the unary logical complement expression is true if the operand value is false and false if the operand value is true.

4.7 Multiplicative Operators

The operators `*`, `/`, and `%` are called the *multiplicative operators*. They have the same precedence and are syntactically left-associative (they group left-to-right).

```
MultiplicativeExpression:
     UnaryExpression
     MultiplicativeExpression * UnaryExpression
     MultiplicativeExpression / UnaryExpression
     MultiplicativeExpression % UnaryExpression
```

The type of each of the operands of a multiplicative operator must be a primitive numeric type or a run-time error occurs.

4.7.1 Multiplication Operator *

The binary `*` operator performs multiplication, producing the product of its operands. Multiplication is commutative. Multiplication is not always associative in JavaScript.

If an integer multiplication overflows, then the result is the low-order bits of the mathematical product as represented in some sufficiently large two's-complement format. As a result, if overflow occurs, then the sign of the result may not be the same as the sign of the mathematical product of the two operand values.

The result of a floating-point multiplication is governed by the rules of IEEE 754 arithmetic:

- If either operand is NaN, the result is NaN.
- If neither operand is NaN, the sign of the result is positive if both operands have the same sign, negative if the operands have different signs.
- Multiplication of an infinity by a zero results in NaN.
- Multiplication of an infinity by a finite value results in a signed infinity. The sign is determined by the rule already stated above.
- In the remaining cases, where neither an infinity or NaN is involved, the product is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, the result is then an infinity of appropriate sign. If the magnitude is too

small to represent, the result is then a zero of appropriate sign. The JavaScript language requires support of gradual underflow as defined by IEEE 754.

4.7.2 Division Operator /

The binary / operator performs division, producing the quotient of its operands. The left-hand operand is the dividend and the right-hand operand is the divisor.

JavaScript does not perform integer division. The operands and result of all division operations are floating-point numbers. The result of division is determined by the specification of IEEE arithmetic:

- If either operand is NaN, the result is NaN.
- If neither operand is NaN, the sign of the result is positive if both operands have the same sign, negative if the operands have different signs.
- Division of an infinity by an infinity results in NaN.
- Division of an infinity by a finite value results in a signed infinity. The sign is determined by the rule already stated above.
- Division of a finite value by an infinity results in a signed zero. The sign is determined by the rule already stated above.
- Division of a zero by a zero results in NaN; division of zero by any other finite value results in a signed zero. The sign is determined by the rule already stated above.
- Division of a non-zero finite value by a zero results in a signed infinity. The sign is determined by the rule already stated above.
- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, the quotient is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent, we say the operation underflows and the result is then a zero of appropriate sign. The JavaScript language requires support of gradual underflow as defined by IEEE 754.

4.7.3 Remainder Operator %

The binary `%` operator is said to yield the remainder of its operands from an implied division; the left-hand operand is the dividend and the right-hand operand is the divisor. In C and C++, the remainder operator accepts only integral operands, but in JavaScript, it also accepts floating-point operands.

The result of a floating-point remainder operation as computed by the `%` operator is *not* the same as the so-called "remainder" operation defined by IEEE 754. The IEEE 754 "remainder" operation computes the remainder from a rounding division, not a truncating division, and so its behavior is *not* analogous to that of the usual integer remainder operator. Instead the JavaScript language defines `%` on floating-point operations to behave in a manner analogous to that of the JavaScript integer remainder operator; this may be compared with the C library function `fmod`.

The result of a JavaScript floating-point remainder operation is determined by the rules of IEEE arithmetic:

- If either operand is NaN, the result is NaN.
- If neither operand is NaN, the sign of the result equals the sign of the dividend.
- If the dividend is an infinity, or the divisor is a zero, or both, the result is NaN.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
- If the dividend is a zero and the divisor is finite, the result equals the dividend.
- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, the floating-point remainder r from a dividend n and a divisor d is defined by the mathematical relation $r = n - (d * q)$ where q is an integer that is negative only if n/d is negative and positive only if n/d is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of n and d.

Examples

5%3 produces 2

5%(-3) produces 2

(-5)%3 produces -2

(-5)%(-3) produces -2

5.2345%3.0 produces 2.2345

5.0%(-3.0) produces 2.0

(-5.0)%3.0 produces -2.0

(-5.0)%(-3.0) produces -2.0

4.8 Additive Operators

The operators `+` and `-` are called the *additive operators*. They have the same precedence and are syntactically left-associative (they group left-to-right).

```
AdditiveExpression:
     MultiplicativeExpression
     AdditiveExpression + MultiplicativeExpression
     AdditiveExpression - MultiplicativeExpression
```

In an *AdditiveExpression*, if either of the operands is of type string, then the other operand is convered to a string and the operation is a string concatenation operation (see 4.8.1 String Concatenation Operator +). If one of the operands is of type `boolean`, then it is converted to a `number` (true becomes 1 and false becomes 0).

4.8.1 String Concatenation Operator +

If only one operand expression is of type `string`, then string conversion is performed on the other operand to produce a string at run time. The result is a reference to a newly created `string` object that is the concatenation of the two operand strings. The characters taken from the left-hand operand precede the characters taken from the right-hand operand in the newly created string.

4.8.1.1 String Conversion

Any type may be converted to type `string` by an invocation of the `toString` method of the referenced object. For more information, see 3.1.2 Type Conversion.

4.8.1.2 Examples of String Concatenation

The example expression:

```
"The square root of 2 is " + Math.sqrt(2)
```

produces the result:

```
"The square root of 2 is 1.4142135623730952"
```

The + operator is syntactically left-associative, no matter whether it is later determined by type analysis to represent string concatenation or addition. In some cases care is required to get the desired result. For example, the expression: `a + b + c` is always regarded as meaning `(a + b) + c`. Therefore the result of the expression: `1 + 2 + " fiddlers"` is `"3 fiddlers"` but the result of `"fiddlers " + 1 + 2` is `"fiddlers 12"`.

4.8.2 Additive Operators (+ and -) for Numeric Types

The binary `+` operator performs addition when applied to two operands of numeric type, producing the sum of the operands. The binary `-` operator performs subtraction, producing the difference of two numeric operands.

Addition is a commutative operation, but not always associative.

If addition overflows, then the result is the low-order bits of the mathematical sum as represented in some sufficiently large two's-complement format. If overflow occurs, then the sign of the result will not be the same as the sign of the mathematical sum of the two operand values.

The result of an addition is determined using the rules of IEEE arithmetic:

- If either operand is NaN, the result is NaN.

- The sum of two infinities of opposite sign is NaN.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and a finite value is equal to the infinite operand.
- The sum of two zeros is zero.
- The sum of a zero and a nonzero finite value is equal to the nonzero operand.
- The sum of two nonzero finite values of the same magnitude and opposite sign is zero.
- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, and the operands have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, the operation overflows and the result is then an infinity of appropriate sign. If the magnitude is too small to represent, the operation underflows and the result is then zero. The JavaScript language requires support of gradual underflow as defined by IEEE 754.

The binary `-` operator performs subtraction when applied to two operands of numeric type producing the difference of its operands; the left-hand operand is the minuend and the right-hand operand is the subtrahend. For both integer and floating-point subtraction, it is always the case that `a-b` produces the same result as `a+(-b)`.

4.9 Shift Operators

The shift operators include the left shift <<, the signed right shift >>, and the unsigned right shift >>>; they are syntactically left-associative (they group left-to- right). The left-hand operand of a shift operator is the value to be shifted; the right-hand operand specifies the shift distance.

```
ShiftExpression:
     AdditiveExpression
     ShiftExpression << AdditiveExpression
     ShiftExpression >> AdditiveExpression
     ShiftExpression >>> AdditiveExpression
```

The type of each of the operands of a shift operator must be convertible to a primitive integral type or a run-time error occurs. At run time, shift operations are performed on the two's complement integer representation of the value of the left operand.

The value of `n<<s` is `n` left-shifted `s` bit positions; this is equivalent (even if overflow occurs) to multiplication by two to the power `s`.

The value of `n>>s` is `n` right-shifted `s` bit positions with sign-extension. For non-negative values of `n`, this is equivalent to truncating integer division, as computed by the integer division operator `/`, by two to the power `s`.

The value of `n>>>s` is `n` right-shifted `s` bit positions with zero-extension. If `n` is positive, the result is the same as that of `n>>s`; if `n` is negative, the result is equal to that of the expression `(n>>s)+(2<<~s)` if the type of the left-hand operand is `int` and to the result of the expression `(n>>s)+(2L<<~s)` if the type of the left-hand operand is `long`. The added term `(2<<~s)` or `(2L<<~s)` cancels out the propagated sign bit. (Note that, because of the implicit masking of the right-hand operand of a shift operator, `~s` as a shift distance is equivalent to `31-s` when shifting an `int` value and to `63-s` when shifting a `long` value!)

4.10 Relational Operators

The relational operators are syntactically left-associative (they group left-to-right), for example, `a<b<c` parses as `(a<b)<c`. The operands must be convertible to numbers, or a run-time error occurs.

```
RelationalExpression:
     ShiftExpression
     RelationalExpression < ShiftExpression
     RelationalExpression > ShiftExpression
     RelationalExpression <= ShiftExpression
     RelationalExpression >= ShiftExpression
```

The type of a relational expression is always `boolean`.

4.10.1 Numerical Comparison Operators

The type of each of the operands of a numerical comparison operator must be convertible to a `number,` or a run-time error occurs. If the operands are both of type string, then they are compared as strings.

The result of a numerical comparison, as determined by the specification of the IEEE 754 standard, is:

- If either operand is NaN, the result is `false`.
- All values other than NaN are ordered, with negative infinity less than all finite values, and positive infinity greater than all finite values.

Subject to these considerations, the following rules then hold for operands other than NaN:

- The value produced by the < operator is `true` if the value of the left-hand operand is less than the value of the right-hand operand, and otherwise is `false`.
- The value produced by the <= operator is `true` if the value of the left-hand operand is less than or equal to the value of the right-hand operand, and otherwise is `false`.
- The value produced by the > operator is `true` if the value of the left-hand operand is greater than the value of the right-hand operand, and otherwise is `false`.
- The value produced by the >= operator is `true` if the value of the left-hand operand is greater than or equal to the value of the right-hand operand, and otherwise is `false`.

4.11 Equality Operators

The equality operators are syntactically left-associative (they group left-to-right), for example, `a==b==c` parses as `(a==b)==c`. The result type of `a==b` is always `boolean`, and the value of c is therefore converted to `boolean` before it is compared. For example, if c is a number, then if it is zero, it has a boolean value of `false`; otherwise it is `true`.

Thus `a==b==c` does *not* test to see whether `a`, `b`, and `c` are all equal.

```
EqualityExpression:
     RelationalExpression
     EqualityExpression == RelationalExpression
     EqualityExpression != RelationalExpression
```

The == (equal to) and the != (not equal to) operators are analogous to the relational operators except for their lower precedence. Thus `a<b==c<d` is `true` whenever `a<b` and `c<d` have the same truth-value.The type of an equality expression is always `boolean`.

In all cases, `a!=b` produces the same result as `!(a==b)`. The equality operators are commutative.

4.11.1 Numerical Equality Operators == and !=

Equality testing is performed in accordance with the rules of the IEEE 754 standard:

- If either operand is NaN, the result of `==` is `false` but the result of `!=` is `true`. Indeed, the test `x!=x` is true if and only if the value of `x` is NaN.
- Otherwise, two distinct numeric values are considered unequal by the equality operators.

Subject to these considerations, the following rules then hold for operands other than NaN:

- The value produced by the `==` operator is `true` if the value of the left-hand operand is equal to the value of the right-hand operand, and otherwise is `false`.

- The value produced by the `!=` operator is `true` if the value of the left-hand operand is not equal to the value of the right-hand operand, and otherwise is `false`.

For example,

```
x = 345
y = 345
z = 1
b1 = 2== 2== 1
b2 = 2 == 3 == 1
b3 = 2 == 3 == 0
b4 = 2 == 3 == false
println("b1= " + b1)
println("b2 = " + b2 )
println("b3 = " + b3 )
println("b4 = " + b4 )
```

The output of this script is

```
b1= true
b2 = false
b3 = true
b4 = true
```

4.11.2 Boolean Equality Operators

If the operands of an equality operator are both of type `boolean`, then the operation is boolean equality. The `boolean` equality operators are associative as well as commutative.

The result of `==` is `true` if the operands are both `true` or both `false`; otherwise the result is `false`.

The result of `!=` is `false` if the operands are both `true` or both `false`; otherwise the result is `true`. Thus `!=` behaves the same as `^` when applied to boolean operands.

4.12 Bitwise and Logical Operators

The bitwise and logical operators include the AND operator &, exclusive OR operator `^`, and inclusive OR operator `|`. These operators have different precedence, with & having the highest precedence and `|` the lowest precedence. Each operator is syntactically left-associative (each groups left-to-right). Each operator is both commutative and associative.

```
AndExpression:
     EqualityExpression
     AndExpression & EqualityExpression

ExclusiveOrExpression:
     AndExpression
     ExclusiveOrExpression ^ AndExpression

InclusiveOrExpression:
     ExclusiveOrExpression
     InclusiveOrExpression | ExclusiveOrExpression
```

The bitwise and logical operators may be used to compare two operands of numeric type or two operands of type `boolean`. All other cases result in a runtime error.

4.12.1 Integer Bitwise Operators

The operands of an operator &, `^`, or `|` are converted to integer values, and the result is also an integer numeric value.

For &, the result value is the bitwise AND of the operand values.

For `^`, the result value is the bitwise exclusive OR of the operand values.

For `|`, the result value is the bitwise inclusive OR of the operand values.

For example, the result of the expression `0xff00` & `0xf0f0` is `0xf000`. The result of `0xff00 ^ 0xf0f0` is `0x0ff0`.The result of `0xff00 | 0xf0f0` is `0xfff0`.

4.12.2 Boolean Logical Operators

When both operands of a &, `^`, or `|` operator are of type `boolean`, then the type of the bitwise operator expression is `boolean`.

For &, the result value is `true` if both operand values are `true`; otherwise the result is `false`. For `^`, the result value is `true` if the operand values are different; otherwise the result is `false`. For `|`, the result value is `false` if both operand values are `false`; otherwise the result is `true`.

4.13 Conditional-And Operator

The && operator is like & but evaluates its right-hand operand only if the value of its left-hand operand is `true`. It is syntactically left-associative (it groups left-to-right). It is fully associative with respect to both side effects and result value; that is, for any expressions *a*, *b*, and *c*, evaluation of the expression ((*a*)&&(*b*))&&(*c*) produces the same result, with the same side effects occurring in the same order, as evaluation of the expression (*a*)&&((*b*)&&(*c*)).

```
ConditionalAndExpression:
     InclusiveOrExpression
     ConditionalAndExpression && InclusiveOrExpression
```

Each operand of && must be convertible to type `boolean` or a run-time error occurs. The type of a conditional-and expression is always `boolean`.

At run time, the left-hand operand expression is evaluated first; if its value is `false`, the value of the conditional-and expression is `false` and the right-hand operand expression is not evaluated. If the value of the left-hand operand is `true`, then the right-hand expression is evaluated and its value becomes the value of the conditional-and expression. Thus && computes the same result as & on `boolean` operands. It differs only in that the right-hand operand expression is evaluated conditionally rather than always.

4.14 Conditional-Or Operator

The `||` operator is like `|` but evaluates its right-hand operand only if the value of its left-hand operand is `false`. It is syntactically left-associative (it groups left-to-right). It is fully associative with respect to both side effects and result value; that is, for any expressions *a*, *b*, and *c*, evaluation of the expression `((`*a*`)||(`*b*`))||(`*c*`)` produces the same result, with the same side effects occurring in the same order, as evaluation of the expression `(`*a*`)||((`*b*`)||(`*c*`))`.

```
ConditionalOrExpression:
     ConditionalAndExpression
     ConditionalOrExpression || ConditionalAndExpression
```

Each operand of `||` must be convertible to type `boolean` or a run-time error occurs. The type of a conditional-or expression is always `boolean`.

At run time, the left-hand operand expression is evaluated first; if its value is `true`, the value of the conditional-or expression is `true` and the right-hand operand expression is not evaluated. If the value of the left-hand operand is `false`, then the right-hand expression is evaluated and its value becomes the value of the conditional-or expression. Thus `||` computes the same result as `|` on `boolean` operands. It differs only in that the right-hand operand expression is evaluated conditionally rather than always.

4.15 Conditional Operator `? :`

The conditional operator `? :` uses the boolean value of one expression to decide which of two other expressions should be evaluated.

The conditional operator is syntactically right-associative (it groups right-to-left), so that `a?b:c?d:e?f:g` means the same as `a?b:(c?d:(e?f:g))`.

```
ConditionalExpression:
     ConditionalOrExpression
     ConditionalOrExpression ? Expression : ConditionalExpression
```

The conditional operator has three operand expressions; `?` appears between the first and second expressions, and `:` appears between the second and third expressions.

The first expression must be of type `boolean` or a run-time error occurs.

The conditional operator may be used to choose between second and third operands of numeric type, or second and third operands of type `boolean`. All other cases result in a run-time error.

Note that it is not permitted for either the second or the third operand expression to be an invocation of a `void` function. In fact, it is not permitted for a conditional expression to appear in any context where an invocation of a `void` function could appear

At run time, the first operand expression of the conditional expression is evaluated first; its `boolean` value is then used to choose either the second or the third operand expression:

- If the value of the first operand is `true`, then the second operand expression is chosen.
- If the value of the first operand is `false`, then the third operand expression is chosen.

The chosen operand expression is then evaluated and the resulting value is converted to the type of the conditional expression as determined by the rules stated above. The operand expression not chosen is not evaluated for that particular evaluation of the conditional expression.

4.16 Assignment Operators

There are twelve assignment operators; all are syntactically right-associative (they group right-to-left). Thus `a=b=c` means `a=(b=c)`, which assigns the value of `c` to `b` and then assigns the value of b to a.

```
AssignmentExpression:
     ConditionalExpression
     Assignment

Assignment:
     LeftHandSide AssignmentOperator AssignmentExpression

LeftHandSide:
     ExpressionName
     identifierName
     ArrayAccess

AssignmentOperator: one of
     = *= /= %= += -= <<= >>= >>>= &= ^= |=
```

The result of the first operand of an assignment operator must be a variable or a run-time error occurs. The type of the assignment expression is the type of the variable.

At run time, the result of the assignment expression is the value of the variable after the assignment has occurred. The result of an assignment expression is not itself a variable.

4.16.1 Simple Assignment Operator =

At run time, the left-hand operand expression is evaluated first, resulting in a variable. Next the right-hand operand is evaluated and the result is stored into the variable.

4.17 Special operators

JavaScript has several operators described in this section.

4.17.1 The new operator

The new operator returns an object created with a constructor. The constructor can be either a constructor function in the current program or one of the built-in constructors:

```
new constructor

constructor:
     ConstructorFunction
     Array
     Date
     Math
     String(StringValue)
     Function(ParameterList, "block")
     Boolean(BooleanValue)
```

For more information on constructor functions, see 5.3 Constructor Functions.

4.17.2 The delete operator

The **delete** operator removes a property definition, frees the memory associated with it, and returns undefined.

```
deleteExpression
     delete(identifierName)
     delete identifierName
```

4.17.3 The typeof operator

The typeof operator returns a string specifying the type of its unevaluated operand. The operand is any expression. Syntax:

```
typeof expression
```

The string returned is one of

- “undefined”
- “object”
- “function”
- “number”
- “boolean”
- “string”

If the operand is a constructor, such as Date, the return value is “function.”

For example,

```
typeof foo returns "undefined"          // where foo is undefined
typeof eval returns "function"
typeof null returns "object"
typeof 3.14 returns "number"
typeof true returns "boolean"
typeof "a string" returns "string"
```

4.17.4 The void operator

The void operator takes an expression of any type as its operand, and returns undefined.

```
voidExpression:
     void expr
```

Chapter

5

5 Object Model

The framework for the JavaScript object model is built around

- constructor functions, that provide the underpinnings of the JavaScript object model.
- Built-in objects, that have pre-defined properties and methods.
- The **new** operator, that enables creation of new objects with specified property values
- object syntax: the way that an object’s properties and methods are accessed as *`object.property`*, *`object.method`*, and *`object`*`[`*`index`*`].`

Formal conceptual framework TBD.

5.1 Functions

A function is a set of statements that performs a specific task. A function is defined by a function *definition* that specifies the function’s name and the statements that it contains. A function is executed by a function *invocation* (also known as a function *call*).

5.1.1 Definition

A function definition provides:

- The name of the function
- The parameters of the function
- The statements performed by the function

The syntax for defining a function is described in 6.4.10 Function Definition Statement. A function must be declared before it is invoked.

5.1.2 Invocation

A function invocation executes the statements in the function and optionally returns a value. It consists of a defined function's name, followed by a parameter list in parentheses.

```
functionInvocation:
     functionName(parameterList_opt)

parameterList:
     identifierName
     identifierName, parameterList
```

The number of arguments in an invocation does not have to match the number of arguments in the function definition. Each argument in the invocation will be matched from left to right with the arguments in the definition. Any argument in the definition for which there is no argument in the invocation will be undefined. If there are more arguments in the invocation than in the definition, then the extra arguments are accessible within the function using the arguments array; see 5.1.4 The arguments array.

The value of a function invocation expression is the value of the expression following the return statement that returned control to the invocation, or undefined if there was no return statement or a return statement without an expression.

Note that JavaScript does not require function arguments to be of any particular type.

5.1.3 The caller property

Every function has a caller property that is a reference to the object or function that invoked a function, if any. It is valid only within a function. The syntax is:

```
functionName.caller
```

It evaluates to the object or function that invoked the function. If the function call was a top-level invocation, that is, not made from within another function or from an object, then caller is null.

Better description TBD.

5.1.4 The arguments array

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

```
functionName.arguments[i]
```

where *functionName* is the name of the function and *i* is the ordinal number of the argument, starting at zero. So, the first argument passed to a function named **myfunc** would be `myfunc.arguments[0]`. As for other arrays in JavaScript, the arguments array has a length property that indicates the number of elements in the array. Thus, within the body of the function, the actual number of arguments passed to a function is accessible as `arguments.length` .

Since a a function can be invoked with more arguments than the number of parameters in its definition, the *arguments* array provides a way to access the excess arguments in the invocation.

5.2 This

The special keyword `this` is used to refer to the current object. The current object is defined as follows:

- in an eval expression, string argument
  What does this mean???

- In a method, it is the object to which the method belongs.
- In a constructor function, it is the object being defined.
- In a top-level script, it is the top-level object defined by the particular JavaScript implementation. For example, in Netscape Navigator, the top-level object is the window object.

5.3 Constructor Functions

A *constructor* is a function used to define new objects. In addition to the constructors for built-in objects, described in Chapter 7, “Built-in Functions and Objects”, user-defined constructor functions create objects of the user’s own definition.

Brendan marked to delete, but shouldn’t we define some specific characteristics of a constructor function?

```
userDefinedConstructor
     function constructorName (parameterList) { memberAssignments }

parameterList:
     identifierName
     identifierName, parameterList

memberAssignments:
     memberAssignment
     memberAssignment propertyAssignment

memberAssignment:
     this.identifierName = value

value:
     identifierName
     Literal
```

Each *memberAssignment* statement defines a *property* or *method* of the object. If the value assigned is an object, string, number, or Boolean, then the *memberAssignment* defines a property. If *identifierName* corresponds to a function name in the current scope, then the *propertyAssignment* defines a method of the object. Frequently, an *identifierName* in a *memberAssignment* will correspond to one of the *identifierNames* in the constructor’s *parameterList*, such that the values to be assigned are passed as parameters to the constructor function.

5.3.1 Associative Arrays

Properties are accessible as elements of an associative array. That is, if there is a property p of an object o, the two expressions

```
o.p
o["p"]
```

are equivalent.

5.3.2 Object prototypes

Every object constructor (including built-in object constructors) has a prototype property. Setting a property of a constructor's prototype property creates a property shared by all objects created by the constructor.

```
prototypeProperty
     constructor.prototype.propertyName
```

Set the value of `constructor.prototype.propertyName` to define a property that is shared by all objects of the specified type, where *constructor* is the name of the constructor function and *propertyName* is the name of the property.

Examples

```
function str_rep(n) {
     var s = "", t = this.toString()
     while (--n >= 0) s += t
     return s
}

function new_rep(n) {
     return "repeat " + this + " " + n + " times."
}

String.prototype.rep = str_rep //add a rep() method to String
s1 = new String("a")
s2 = new String("b")
println(s1.rep(3))
println(s2.rep(7))
```

The output of this example is:

```
aaa
bbbbbbb
```

5.3.3 Defining methods

A *method* is a function associated with an object. A function can be so associated in two ways:

- by the object constructor function, in which case the method belongs to all objects created with the constructor.
- by an assignment statement to assign the method to an individual object instance.

You can define methods for an object type by including a method definition in the object constructor function.

The following syntax associates a function with an existing object:

```
object.methodName = functionName
```

where *object* is an existing object, *methodname* is the name being assigned to the method, and *function_name* is the name of the function.

The method can then be called in the context of the object as follows:

```
object.methodName(parameterList)
```

Example

To format and display the properties of the previously-defined Car objects, the user would define a method called displayCar based on a function called display by modifying the constructor function as follows:

```
function car(make, model, year, owner) {
     this.make = make
     this.model = model
     this.year = year
     this.owner = owner
     this.displayCar = display
}
```

For example, this function might look like this:

```
function displayCar() {
     println("A Beautiful " + this.year + " " + this.make
     + " " + this.model)
}
```

Then the **displayCar** method can be called for each of the objects as follows:

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

5.4 Object Creation

A new object instance is created by using the **new** operator with a constructor, either one of the built-in object constructors described in Chapter 7, “Built-in Functions and Objects”, or a user-defined constructor function, described in 5.3 Constructor Functions.

```
constructor:
     DateConstructor
     ArrayConstructor
     StringConstructor
     BooleanConstructor
     NumberConstructor
     UserDefinedConstructor
```

Example

The following user-defined constructor functions create a Person object with member properties name, age, and sex and a Car object with member properties make, model, year, and owner. Then, two Person objects are created with the new operator, and two Car objects are created. Notice the use of the objects john and fred as arguments to the Car constructor functions. This is an example of members of object type.

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

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

john = new Person("John", 33, "M")
fred = new Person("Fred", 39, "M")

car1 = new Car("Eagle", "Talon TSi", 1993, john)
car2 = new Car("Nissan", "300ZX", 1992, fred)
```

Chapter

6

6 Blocks and Statements

The sequence of execution of a JavaScript program is controlled by *statements*, which are executed for their effect and do not have values.

The first section of this chapter discusses the distinction between normal and abrupt completion of statements. Most of the remaining sections explain the various kinds of statements, describing in detail both their normal behavior and any special treatment of abrupt completion.

6.1 Normal and Abrupt Completion of Statements

Every statement has a normal mode of execution in which certain computational steps are carried out. The following sections describe the normal mode of execution for each kind of statement. If all the steps are carried out as described, the statement is said to *complete normally*. However, the `break`, `continue`, and `return` statements cause a transfer of control that may prevent normal completion of statements that contain them.

If such an event occurs, then execution of one or more statements may be terminated before it completes normally; such statements are said to *complete abruptly*.

Unless otherwise specified, abrupt completion of a substatement causes the immediate abrupt completion of the statement itself, and all succeeding steps in the normal mode of execution are not performed. Unless otherwise specified, a statement completes normally if all substatements it executes complete normally.

6.2 Blocks

A *block* is a sequence of statements and variable declarations statements within braces.

```
Block:
     Statement
     { BlockStatements_opt }

BlockStatements:
     BlockStatement
     BlockStatements BlockStatement

BlockStatement:
     VariableDeclarationStatement
     Statement
```

A block is executed by executing each of the variable declarations and statements in order from first to last (left to right). If all of these block statements complete normally, then the block completes normally. If any of these block statements complete abruptly for any reason, then the block completes abruptly.

6.3 Variable Declaration Statements

Variables can be declared two ways in JavaScript:

- By use: JavaScript recognizes a new variable and declares it automatically.
- By a declaration statement using the **var** keyword.

A variable declaration statement declares one or more variable names.

```
VariableDeclarationStatement:
     VariableDeclaration ;

VariableDeclaration:
     var VariableDeclarators
```

The following productions are repeated here for clarity:

```
VariableDeclarators:
     VariableDeclarator
     VariableDeclarators , VariableDeclarator

VariableDeclarator:
     Identifier
     Identifier = Expression
```

A variable declaration can also appear in the header of a `for` statement. In this case it is executed in the same manner as if it were part of a variable declaration statement.

Each declarator in a local variable declaration declares one local variable, whose name is the *Identifier* that appears in the declarator. If the variable is assigned a value, the variable assumes the type compatible with that value, otherwise it is undefined.

The scope of variable declarations is described in 3.5.1 Declarations and Scoping.

6.4 Statements

Some of the statements in the JavaScript language correspond to statements in Java, but some are unique to JavaScript.

```
Statement:
     EmptyStatement
     IfThenStatement
     WhileStatement
     ForStatement
     BreakStatement
     ContinueStatement
     ReturnStatement
     WithStatement
     ForInStatement
```

6.4.1 The Empty Statement

An empty statement does nothing.

```
EmptyStatement:
     ;
```

Execution of an empty statement always completes normally.

6.4.2 The if Statement

The `if` statement allows conditional execution of a statement or a conditional choice of two statements, executing one or the other but not both.

```
IfThenStatement:
     if ( Expression ) Block
     if ( Expression ) Block else Block
```

The *Expression* must be defined, or a run-time error occurs. The *Expression* is evaluated:

- If the value of *Expression* is `true`, then the first *Block* is executed.
- If the value of *Expression* is `false`, and there is no `else` clause, then no further action is taken. If there is an `else` clause, then the *Block* after the `else` keyword is executed.

Because JavaScript parses top-down, it does not encounter any ambiguity. in this statement:

```
if condition1
     if Ccndition2
          Statement1
     else
          Statement2
```

The JavaScript top-down parser can determine that the `else` goes with the second `if` statement to form an `if-else`.

6.4.3 The while Statement

The `while` statement executes an *Expression* and a *Statement* repeatedly until the value of the *Expression* is `false`.

```
WhileStatement:
     while ( Expression ) Block
```

The *Expression* must be defined, or a run-time error occurs.

A `while` statement is executed by first evaluating the *Expression*:

- If the value is `true`, then the *Block* is executed. Then there is a choice:
  - If execution of the *Block* completed normally, then the entire `while` statement is executed again, beginning by re-evaluating the *Expression*.
  - If execution of the *Block* completed abruptly, see below.
- If the value of the *Expression* is `false`, no further action is taken and the `while` statement completes normally.

If the value of the *Expression* is `false` the first time it is evaluated, then the *Block* is not executed.

6.4.3.1 Abrupt Completion

Abrupt completion of the contained *Block* is handled in the following manner:

- If execution of the *Block* completed abruptly because of a `break` statement, no further action is taken and the `while` statement completes normally.
- If execution of the *Block* completed abruptly because of a `continue` statement, then the entire `while` statement is executed again.
- If execution of the *Block* completed abruptly because of a `return`, the `while` statement completes abruptly for the same reason.

6.4.4 The for Statement

The `for` statement executes some initialization code, then executes an *Expression*, a *Block*, and some update code repeatedly until the value of the Expression is `false`.

```
ForStatement:
     for ( ForInit_opt ; Expression_opt ; ForUpdate_opt ) Block

ForInit:
     StatementExpressionList
     VariableDeclaration

ForUpdate:
     StatementExpressionList
```

```
StatementExpressionList:
     StatementExpression
     StatementExpressionList , StatementExpression
```

The *Expression* must be defined, or a run-time error occurs.

6.4.4.1 Initialization

A `for` statement is executed by first executing the *ForInit* code:

- If the *ForInit* code is a list of statement expressions, the expressions are evaluated in sequence from left to right; their values, if any, are discarded.
- If the *ForInit* code is a variable declaration, it is executed as if it were a variable declaration statement appearing in a block. In this case, the scope of a declared variable is its own initializer, any further declarators in the *ForInit* part, plus the *Expression*, *ForUpdate*, and contained *Statement* of the `for` statement.
- If the *ForInit* part is not present, no action is taken.

6.4.4.2 Iteration

Next a `for` iteration step is performed, as follows:

- If the *Expression* is present, it is evaluated, and there is then a choice based on the presence or absence of the *Expression* and the resulting value if the *Expression* is present:
- If the *Expression* is not present, or it is present and the value resulting from its evaluation is `true`, then the contained *Block* is executed. Then there is a choice:
  - If execution of the *Block* completed normally, then the following two steps are performed in sequence:
  - — First, if the *ForUpdate* part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If the *ForUpdate* part is not present, no action is taken.
  - — Second, another `for` iteration step is performed.
  - If execution of the *Block* completed abruptly, see below.

- If the *Expression* is present and the value resulting from its evaluation is `false`, no further action is taken and the `for` statement completes normally.

If the value of the *Expression* is `false` the first time it is evaluated, then the *Block* is not executed.

If the *Expression* is not present, then the `for` statement cannot complete normally; only abrupt completion (such as use of a `break` statement) can terminate its execution.

6.4.4.3 Abrupt Completion

Abrupt completion of the contained *Block* is handled in the following manner:

- If execution of the *Block* completed abruptly because of a `break` statement, no further action is taken and the `for` statement completes normally.
- If execution of the *Block* completed abruptly because of a `continue` statement, then the following two steps are performed in sequence:
  - — First, if the *ForUpdate* part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If the *ForUpdate* part is not present, no action is taken.
  - — Second, another `for` iteration step is performed.
- If execution of the *Block* completed abruptly for any other reason, the `for` statement completes abruptly for the same reason.

6.4.5 The break Statement

The break statement transfers control out of an enclosing statement.

```
BreakStatement:
     break
```

A `break` statement transfers control to the innermost enclosing `while` or `for` statement; this statement, called the *break target*, then immediately completes normally. If no `while` or `for` statement encloses the `break` statement, a run-time error occurs. A `break` statement always completes abruptly.

Example TBD.

6.4.6 The continue Statement

The `continue` statement may occur only in an `while` or `for` statement, known as an *iteration statement*. Control passes to the loop-continuation point of an iteration statement.

```
ContinueStatement:
     continue
```

A `continue` statement transfers control to the innermost enclosing `while` or `for` statement; this statement, which is called the *continue target*, then immediately ends the current iteration and begins a new one. If no `while` or `for` statement encloses the `continue` statement, a run-time error occurs. A `continue` statement always completes abruptly.

Example TBD.

6.4.7 The return Statement

The `return` statement returns control to the invoker of a function.

```
ReturnStatement:
     return Expressionopt
```

A `return` statement with an *Expression* must be contained in a function definition or a run-time error occurs.

A `return` statement with an *Expression* transfers control to the invoker of the function; the value of the *Expression* becomes the value of the funciton invocation.

6.4.8 The with Statement

The `with` statement establishes the default object for a set of statements. Within the set of statements, any property references that do not specify an object are assumed to be for the default object.

```
withStatement:
     with (object) block
```

object specifies the default object to use for the *statements*. The parentheses around *object* are required.
statements is any block of statements.

When `with` statements are nested, the outermost statement defines the default top-level object; the next with statement defines the next level object, and so on.

Better description of scoping TBD.

Example

The following **with** statement specifies that the Math object is the default object. The statements following the **with** statement refer to the PI property and the cos and sin methods, without specifying an object. JavaScript assumes the Math object for these references.

```
var a, x, y
var r=10
with (Math) {
   a = PI * r * r
   x = r * cos(PI)
   y = r * sin(PI/2)
}
```

6.4.9 The 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 in the *block*.

```
ForInStatement:
     for (Identifier in ObjectReference) Block
```

ObjectReference is an identifier that is the object for which the properties are iterated.

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. This example accesses object properties as elements of an associative array, as described in 5.3.1 Associative Arrays.

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

6.4.10 Function Definition Statement

A function definition declares a JavaScript function *name* with the specified parameters.

```
functionDeclaration:
     function name(parameterList) block

parameterList:
     identifierName
     identifierName, parameterList
```

A function declaration statement cannot be nested inside another a function declaration statement or any other statement.

Primitive values are passed to functions *by value*. In other words, 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.

Example

```
function factorial(n) {
     if (n<=1)
          return 1
     else
          return (n * factorial(n-1) )
}
```

Chapter 7

7 Built-in Functions and Objects

JavaScript also has several “top-level” built-in functions. JavaScript also has four built-in objects: Array, Date, Math, and String. Each object has special-purpose properties and methods associated with it. JavaScript also has constructors for objects corresponding to the Boolean and numeric primitive types.

7.1 Built-in functions

JavaScript has five “top-level” functions built in to the language. They are eval, parseInt, parseFloat, escape, and unescape.

7.1.1 eval

Note: Need discussion of eval as method of other objects, eg. eval in scope of an object.

Evaluates a string and returns a value.

```
eval(string)
```

string is a string-valued expression.

If the string argument represents a numeric or Boolean expression, eval evaluates the expression. If the argument represents one or more JavaScript statements, eval performs the statements. Otherwise, it generates a run-time error.

The scope of execution is determined as follows:

- If the call to eval is preceded by a period and an object reference, this object is the scope.
- If there is no object to the left, then the scope is the object for which eval is defined.
- Otherwise, the scope is the innermost default scope (see 3.5.1 Declarations and Scoping).

Clarification TBD.

Examples

Both of the println statements below display 42. The first evaluates the string “x + y + 1,” and the second evaluates the string “42.”

```
var x = 2
var y = 39
var z = "42"
println(eval("x + y + 1"))
println(eval(z))
```

In the following example, the getFieldName function returns a string value that may represent a number or string. The second statement below uses eval to display the value of the form element.

```
var field = getFieldName(3)
println("Field named ", field, " has value of ", eval(field + ".value"))
```

The following example uses eval to evaluate the string *str*. This string consists of JavaScript statements that do different things, depending on the value of x. When the second statement is executed, eval will cause these statements to be performed, and it will also evaluate the set of statements and return the value that is assigned to z.

```
var str = "if (x == 5) {z = 42; println("z is" + z);} else z = 0; "
println("z is " + eval(str))
```

7.1.2 parseInt

Parses a string argument and returns an integer of the specified radix or base. Syntax:

```
parseInt(string)
parseInt(string, radix)
```

string is a string that represents the value you want to parse.
radix is an integer that represents the radix of the return value.

The **parseInt** function parses its first argument, a string, and attempts to return an integer of the specified radix (base). 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 ninr. For example, for hexadecimal numbers (base sixteen), A through F are used. If a radixes above 36 is specified, **parseInt** returns “NaN.”

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. **parseInt** truncates numbers to integer values.

If the radix is not specified or is specified as zero, JavaScript assumes the following:

- If the input *string* begins with “0x,” the radix is sixteen (hexadecimal).
- If the input *string* begins with “0,” the radix is eight (octal).
- If the input *string* begins with any other value, the radix is ten (decimal).

If the first character cannot be converted to a number, **parseFloat** returns “NaN”.

For example, the following examples all return fifteen:

```
parseInt("F", 16)
parseInt("17", 8)
parseInt("15", 10)
parseInt(15.99, 10)
parseInt("FXX123", 16)
parseInt("1111", 2)
parseInt("15*3", 10)
```

The following examples all return “NaN” or zero:

```
parseInt("Hello", 8)
parseInt("0x7", 10)
parseInt("FFF", 10)
```

Even though the radix is specified differently, the following examples all return seventeen because the input *string* begins with “0x.”

```
parseInt("0x11", 16)
parseInt("0x11", 0)
parseInt("0x11")
```

7.1.3 parseFloat

Parses a string argument and returns a floating point number. Syntax:

```
parseFloat(string)
```

string is a String object or literal.

parseFloat parses its argument, a string, and returns a floating point number. If it encounters a character other than a sign (+ or -), 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, **parseFloat** returns “NaN”.

You can call the **isNaN** function to determine if the result of **parseFloat** is “NaN.” If “NaN” is passed on to arithmetic operations, the operation results will also be “NaN.”

For example, the following examples all return 3.14:

```
parseFloat("3.14")
parseFloat("314e-2")
parseFloat("0.0314E+2")
var x = "3.14"
parseFloat(x)
```

The following example returns “NaN”:

```
parseFloat("FF2")
```

7.1.4 escape

Returns the hexadecimal encoding of an argument in the ISO Latin-1 character set. Syntax:

```
escape("string")
```

string is a string in the ISO Latin-1 character set.

The value returned by the escape function is one of the following:

- For alphanumeric characters, the same character (i.e. the function has no effect).
- For the space character, a + sign.
- For non-alphanumeric characters other than space, a string of the form "%*xx*," where *xx* is the hexadecimal encoding of the ASCII character in the ISO Latin-1 character set.

For example, the following returns "abc%26%25":

```
escape("abc&%")
```

7.1.5 unescape

Returns the ASCII string for the specified value. Syntax:

```
unescape("string")
```

string is a String object or literal.

For each distinct set of characters in the argument string of the following form

- "%*integer*", where *integer* is a number between 0 and 255 (decimal)
- "*hex*", where *hex* is a number between 0x0 and 0xFF (hexadecimal)

unescape returns the corresponding ASCII character in the ISO Latin-1 character set. For characters not in the above form, unescape returns the characters unmodified; except for the + character, for which a space is returned.

For example, the following returns "&":

```
unescape("%26")
```

The following example returns “ab!#”:

```
unescape("ab%21%23")
```

7.2 Array Object

JavaScript arrays are a special kind of object, and are created dynamically. An array object contains a number of variables. The number of variables may be zero, in which case the array is said to be empty. The variables contained in an array have no names; instead they are referenced by array access expressions that use nonnegative integer index values. These variables are called the *components* of the array. If an array has *n* components, we say *n* is the *length* of the array; the components of the array are referenced using integer indices from 0 to *n*-1, inclusive.

Unlike Java, the components of an array do not neccessarily have the same type. An array component can itself be an array, to create essentially multi-dimensional arrays. If, starting from any array type, one considers its component type, and then (if that is also an array type) the component type of that type, and so on, eventually one must reach a component type that is not an array type; the components at this level of the data structure are called the *elements* of the original array.

7.2.1 Constructors

To create an Array object:

```
ArrayConstructor:
     new Array()
     new Array(arrayLength)
     new Array(componentList)

componentList:
     componentValue, componentList
     componentValue

componentValue:
     Identifier
     Literal
```

IdentifierName is an identifier that is the name of the new Array object.

arrayLength is a positive integer-valued numeric expression that specifies the initial length of the Array and becomes the value of the Array object's length property. An *arrayLength* specified to be zero or less results in a run-time error. An *arrayLength* that is not an integer is truncated to an integer.

For example,

```
a = new Array("alpha", "beta", "gamma", "delta")
b = new Array("a", "b", "c", "d")
matrix = new Array(a, b)
for (i = 0; i < a1.length; i++) {
     for (j = 0; j < a1[i].length; j++) {
     println(a1[i][j])
     }
}
```

The ouput of this script is:

```
alpha
beta
gamma
delta
a
b
c
d
```

7.2.2 Properties

An Array object has one property, *length*.

7.2.2.1 length

The *length* property indicates the number of components in an Array object. See the definition of components versus elements in section 1.1. The syntax is:

```
arrayObject.length
```

arrayObject is an Array object.

7.2.3 Methods

The Array object has three methods:

- join: Joins all elements of an array into a string.
- reverse: Reverses elements of an array
- sort: Sorts elements of an array based on a specified comparison function.

7.2.3.1 join

Returns a string containing all the elements of the array. Syntax:

```
arrayName.join(separator)
```

arrayName is the name of an Array object.
separator specifies a string to separate each element of the array. The separator is converted to a string if necessary. If omitted, a comma (,) is used by default.

7.2.3.2 reverse

Reverses the elements of an array: the first array element becomes the last and the last becomes the first. Returns ? The effect of this method is to change the calling object.

Syntax:

```
arrayName.reverse()
```

arrayName is the name of an Array object.

7.2.3.3 sort

Sorts the elements of an array. Syntax:

```
arrayName.sort(compareFunction)
arrayName.sort()
```

arrayName is the name of an Array object.
compareFunction is the name of a function that defines the sort order. It must be a function defined in the current program, a method of a built-in object, or a built-in function.

If omitted, the array is sorted lexicographically (in dictionary order) according to the string conversion of each element.

Strict definition of sort order for ASCII character set TBD.

7.3 Boolean Object

The Boolean object represents a primitive `boolean` value.

7.3.1 Constructors

The Boolean constructor creates an object with a Boolean value.

```
BooleanConstructor:
     new Boolean(BooleanLiteral)
     new Boolean()
```

If no argument is provided, then the constructor creates a object with Boolean value `false`.

7.3.2 Properties

The Boolean object has no properties.

7.3.3 Methods

The Boolean object has toString and valueOf methods.

7.4 Date Object

The Date object provides a system-independent abstraction of dates and times. Dates may be constructed from a year, month, day of the month, hour, minute, and second, and those six components, as well as the day of the week, may be extracted from a date. Dates may also be compared and converted to a readable string form. A Date is represented to a precision of one millisecond.

The way JavaScript handles dates is very similar to the way Java handles dates: both languages have many of the same date methods, and both store dates internally as the number of milliseconds since January 1, 1970 00:00:00. Dates prior to 1970 are not allowed.

7.4.1 Constructors

There are five forms of a Date constructor:

```
DateConstructor:
     new Date()
     new Date(StringDate)
     new Date(year, month, day)
     new Date(year, month, day, hours, minutes, seconds)
     new Date(year, month, day, hours, minutes)
     new Date(year, month, day, hours)
     new Date(IntegerLiteral)
```

year, *month*, *day*, *hours*, *minutes*, and *seconds* are integers of the format described below.

StringDate is a string representing a date in one of the following forms:

```
month day, year
month day, year, hours:minutes:seconds
month day, year, hours:minutes
month day, year, hours
day month, year
day month, year hours:minutes:seconds
day month, year hours:minutes
day month, year hours
month/day/year
```

year is the year, A.D., or the last two digits of the year*;* *month* is the full name of the month or a three-letter abbreviation, *day* is an integer value for the day of the month; *hours* is an integer between zero and 23; *minutes* and *seconds* are integers between zero and 59. If *hours*, *minutes*, or *seconds* are not specified, then the corresponding value is set to zero.

msSinceEpoch is an integer representing the number of milliseconds since the epoch (00:00:00 GMT on January 1, 1970).

The constructor with no parameters initializes a newly created Date object representing the instant of time that it was created, measured to the nearest millisecond.

Examples

The following examples show several ways to assign dates:

```
today = new Date()
birthday = new Date("December 17, 1995 03:24:00")
birthday = new Date(95,12,17)
birthday = new Date(95,12,17,3,24,0)
```

7.4.2 Properties

The Date object has no pre-defined properties.

7.4.3 Methods

The *Date* object has two kinds of methods: static methods used as member functions of the Date constructor itself, and dynamic methods used as member functions of instances of the Date object.

The static methods are parse and UTC, with syntax:

```
Date.UTC(parameters)
Date.parse(parameters)
```

The syntax for dynamic *Date* methods is:

```
dateObjectName.methodName(parameters)
```

where *dateObjectName* is a Date object created with one of the constructors from 1.2.1.

7.4.3.1 parse

Returns the number of milliseconds in a date string since January 1, 1970, 00:00:00, local time. The syntax is:

```
Date.parse(dateString)
```

dateString is a string value representing a date.

Given a string representing a time, parse returns the time value. It accepts the IETF standard date syntax: “Mon, 25 Dec 1995 13:30:00 GMT.” It understands the continental US time-zone abbreviations, but for general use, use a time-zone offset, for example, “Mon, 25 Dec 1995 13:30:00 GMT+0430” (4 hours, 30 minutes west of the Greenwich meridian). If you do not specify a time zone, the local time zone is assumed. GMT and UTC are considered equivalent.

Because the parse function is a static method of *Date*, you always use it as `Date.parse()`, rather than as a method of a *Date* object you created.

For example, If *IPOdate* is an existing Date object, then

```
IPOdate.setTime(Date.parse("Aug 9, 1995"))
```

7.4.3.2 setDate

Sets the day of the month for a specified date.

```
dateObjectName.setDate(dayValue)
```

dateObjectName is the name of a *Date* object.
dayValue is an integer from one to thirty-one, representing the day of the month.

For example, the second statement below changes the day for *theBigDay* to the 24th of July from its original value.

```
theBigDay = new Date("July 27, 1962 23:30:00")
theBigDay.setDate(24)
```

7.4.3.3 setHours

Sets the hours for a specified date.

```
dateObjectName.setHours(hoursValue)
```

dateObjectName is the name of a *Date* object.
hoursValue is an integer between zero and twenty-three, representing the hour.

For example, the following sets the hour of the Date object theBigDay to 7:

```
theBigDay.setHours(7)
```

7.4.3.4 setMinutes

Sets the minutes for a specified date.

```
dateObjectName.setMinutes(minutesValue)
```

dateObjectName is the name of a *Date* object.
minutesValue is an integer between zero and fifty-nine, representing the minutes.

```
theBigDay.setMinutes(45)
```

7.4.3.5 setMonth

Sets the month for a specified date.

```
dateObjectName.setMonth(monthValue)
```

dateObjectName is the name of a *Date* object.
monthValue is an integer between zero and eleven (representing the months January through December).

For example, the following sets the month of the Date object theBigDay to 6:

```
theBigDay.setMonth(6)
```

7.4.3.6 setSeconds

Sets the seconds for a specified date.

```
dateObjectName.setSeconds(secondsValue)
```

dateObjectName is the name of a *Date* object.
secondsValue is an integer between zero and fifty-nine.

For example, the following sets the seconds of the Date object theBigDay to 30:

```
theBigDay.setSeconds(30)
```

7.4.3.7 setTime

Sets the value of a *Date* object.

```
dateObjectName.setTime(timevalue)
```

dateObjectName is the name of a *Date* object.
timevalue is an integer representing the number of milliseconds since the epoch (1 January 1970 00:00:00).

Use the setTime method to help assign a date and time to another *Date* object.

For example, the following statements set the value of the Date object sameAsBigDay to have the value of the Date object theBigDay:

```
theBigDay = new Date("July 1, 1999")
sameAsBigDay = new Date()
sameAsBigDay.setTime(theBigDay.getTime())
```

7.4.3.8 setYear

Sets the year for a specified date.

```
dateObjectName.setYear(yearValue)
```

dateObjectName is the name of a *Date* object.
yearValue is an integer greater than 1900.

For example, the following sets the year of the Date object theBigDay to 1996:

```
theBigDay.setYear(96)
```

7.4.3.9 toGMTString

Converts a date to a string, using the Internet GMT conventions.

```
dateObjectName.toGMTString()
```

dateObjectName is the name of a *Date* object .

The exact format of the value returned by toGMTString varies according to the platform.

Need xp definition.

In the following example, *today* is a *Date* object:

```
today.toGMTString()
```

In this example, the **toGMTString** method converts the date to GMT (UTC) using the operating system's time-zone offset and returns a string value that is similar to the following form. The exact format depends on the platform.

```
Mon, 18 Dec 1995 17:28:35 GMT
```

7.4.3.10 toLocaleString

Converts a date to a string, using the current locale's conventions.

```
dateObjectName.toLocaleString()
```

dateObjectName is either the name of a *Date* object.

In the following example, *today* is a *Date* object:

```
today.toLocaleString()
```

In this example, **toLocaleString** returns a string value that is similar to the following form. The exact format depends on the platform.

```
12/18/95 17:28:35
```

7.4.3.11 UTC

Returns the number of milliseconds in a *Date* object since January 1, 1970, 00:00:00, Universal Coordinated Time (GMT).

```
Date.UTC(year, month, day [, hrs] [, min] [, sec])
```

year is a year after 1900.
month is a month between zero and eleven.
date is a day of the month between one and thirty-one.
hrs is hours between zero and twenty-three.
min is minutes between zero and fifty-nine.
sec is seconds between zero and fifty-nine.

UTC takes comma-delimited date parameters and returns the number of milliseconds since January 1, 1970, 00:00:00, Universal Coordinated Time (GMT).

Because UTC is a static method of *Date*, you always use it as `Date.UTC()`, rather than as a method of a *Date* object you created.

For example, the following statement creates a *Date* object using GMT instead of local time:

```
gmtDate = new Date(Date.UTC(96, 11, 1, 0, 0, 0))
```

7.5 Math Object

The built-in *Math* object has properties and methods for mathematical constants and functions, respectively.

7.5.1 Constructors

The Math object does not have any constructors. All of its methods and properties are static; that is, they are member functions of the Math object itself. There is no way to create an instance of the Math object.

7.5.2 Properties

The Math object's properties represent mathematical constants. For example, the Math object's *PI* property has the value of pi (3.141...), expressed as

```
Math.PI
```

All properties of Math are read-only values; they cannot be set.

7.5.2.1 E

Euler's constant and the base of natural logarithms, 2.718281828459045. Syntax:

```
Math.E
```

7.5.2.2 LN2

The natural logarithm of two, 0.6931471805599453. Syntax:

```
Math.LN2
```

7.5.2.3 LN10

The natural logarithm of ten, 2.302585092994046. Syntax:

```
Math.LN10
```

7.5.2.4 LOG2E

The base 2 logarithm of e, 1.4426950408889634. Syntax:

```
Math.LOG2E
```

7.5.2.5 LOG10E

The base 10 logarithm of e, 0.4342944819032518. Syntax:

```
Math.LOG10E
```

7.5.2.6 PI

The ratio of the circumference of a circle to its diameter, 3.141592653589793. Syntax:

```
Math.PI
```

7.5.2.7 SQRT1_2

The square root of one-half; equivalently, one over the square root of two, 0.7071067811865476. Syntax:

```
Math.SQRT1_2
```

7.5.2.8 SQRT2

The square root of two, 1.4142135623730951. Syntax:

```
Math.SQRT2
```

7.5.3 Methods

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)
```

7.5.3.1 abs

Returns the absolute value of a number. Syntax:

```
Math.abs(number)
```

number is any numeric expression.

7.5.3.2 acos

Returns the arc cosine (in radians) of a number. Syntax:

```
Math.acos(number)
```

number is a numeric expression between -1 and 1, inclusive.

The **acos** method returns a numeric value between zero and pi radians. If the value of *number* is outside this range, it returns zero.

7.5.3.3 asin

Returns the arc sine (in radians) of a number. Syntax:

```
Math.asin(number)
```

number is a numeric expression with a value between -1 and 1, inclusive

The **asin** method returns a numeric value between -pi/2 and pi/2 radians. If the value of *number* is outside this range, it returns zero.

7.5.3.4 atan

Returns the arc tangent (in radians) of a number.

```
Math.atan(number)
```

number is a numeric expression representing the tangent of an angle.

The **atan** method returns a numeric value between -pi/2 and pi/2 radians.

7.5.3.5 atan2

Returns the angle (*theta* component) of the polar coordinate (*r*,*theta*) that corresponds to the cartesian coordinate specified by the arguments. Syntax:

```
Math.atan2(xCoord, yCoord)
```

xCoord is a numeric expression representing a cartesian x-coordinate.
yCoord is a numeric expression representing a cartesian y-coordinate.

7.5.3.6 ceil

Returns the least integer greater than or equal to its argument.

```
Math.ceil(number)
```

number is any numeric expression.

7.5.3.7 cos

Returns the cosine of a number.

```
Math.cos(number)
```

number is a numeric expression representing the size of an angle in radians.

The **cos** method returns a numeric value between -1 and one, which represents the cosine of the argument.

7.5.3.8 exp

Returns e^{number}, where *number* is the argument, and *e* is Euler's constant, the base of the natural logarithms.

```
Math.exp(number)
```

number is any numeric expression.

7.5.3.9 log

Returns the natural logarithm (base *e*) of a number.

```
Math.log(number)
```

number is any positive numeric expression.

If the value of *number* is outside the suggested range, log returns -1.797693134862316e+308.

7.5.3.10 max

Returns the greater of two numbers. Syntax:

```
Math.max(number1, number2)
```

number1 and *number2* are any numeric arguments or the properties of existing objects.

7.5.3.11 min

Returns the lesser of two numbers. Syntax:

```
Math.min(number1, number2)
```

number1 and *number2* are any numeric arguments or the properties of existing objects.

7.5.3.12 pow

Returns *base* to the *exponent* power, that is, $base^{exponent}$. Syntax:

```
Math.pow(base, exponent)
```

base is any numeric expression.
exponent is any numeric expression.

7.5.3.13 random

Returns a pseudo-random number between zero and one. This method does not have any parameters. Syntax:

```
Math.random()
```

7.5.3.14 round

Returns the value of a number rounded to the nearest integer. Syntax:

```
Math.round(number)
```

number is any numeric expression.

If the fractional portion of *number* is .5 or greater, the argument is rounded to the next highest integer. If the fractional portion of *number* is less than .5, the argument is rounded to the next lowest integer.

7.5.3.15 sin

Returns the sine of a number. Syntax:

```
Math.sin(number)
```

number is a numeric expression, representing the size of an angle in radians.

The **sin** method returns a numeric value between -1 and one, which represents the sine of the argument.

7.5.3.16 sqrt

Returns the square root of a number. Syntax:

```
Math.sqrt(number)
```

number is any non-negative numeric expression. If the value of *number* is outside the required range, sqrt returns zero.

7.5.3.17 tan

Returns the tangent of a number. Syntax:

```
Math.tan(number)
```

number is a numeric expression representing an angle in radians.

7.6 Number Object

The Boolean object corresponds to the `number` primitive type.

7.6.1 Constructors

The Number constructor creates an object with a numeric value.

```
NumberConstructor:
     new Number(IntegerLiteral
     new Number(FloatingPointLiteral)
     new Number()
```

If no argument is provided, the constructor creates an object with numeric value `0`.

7.6.2 Properties

The properties of the Number object are constants.

7.6.2.1 MAX_VALUE

The largest number representable in JavaScript, 1.7976931348623157e308.

7.6.2.2 MIN_VALUE

The smallest number representable in JavaScript, 2.2250738585072014e-308.

7.6.2.3 NaN

The literal NaN, representing a value that is “not a number.”

7.6.3 Methods

The Number object has toString and valueOf methods.

7.7 String Object

A String is an object representing a series of characters.

7.7.1 Constructors

A string object is created whenever a string literal is used or assigned to a variable or with the explicit constructor:

```
identifierName = new String(stringValue)
```

stringValue can be a string literal or string-valued variable.

7.7.2 Properties

A String object has one property, *length*.

7.7.2.1 length

The *length* property indicates the total number of characters in a String object. The syntax is:

```
stringObject.length
```

stringObject is a String object.

For example, the expression

```
mystring = “Hello, World!”
x = mystring.length
```

assigns a value of thirteen to x, because “Hello, World!” has thirteen characters.

7.7.3 Methods

To use String methods:

```
stringName.methodName(parameters)
```

stringName is a String object..
methodName is a method of String.
parameters are the parameters required by the method, if any.

7.7.3.1 indexOf

Returns the index within the calling *string* object of the first occurrence of the specified value, starting the search at *fromIndex*.

```
stringName.indexOf(searchValue)
stringName.indexOf(searchValue, fromIndex)
```

stringName is any string.
searchValue is a string, representing the value to search for.
fromIndex is the location within the calling string to start the search from. It can be any integer from zero to *stringName*.length - 1.

Characters in a string are indexed from left to right. The index of the first character is zero, and the index of the last character is *stringName*.length - 1.

If you do not specify a value for *fromIndex*, JavaScript assumes zero by default. If *searchValue* is not found, JavaScript returns -1.

7.7.3.2 lastIndexOf

Returns the index within the calling *string* object of the last occurrence of the specified value. The calling string is searched backward, starting at *fromIndex*.

```
stringName.lastIndexOf(searchValue,)
stringName.lastIndexOf(searchValue, fromIndex)
```

stringName is any string.
searchValue is a string, representing the value to search for.
fromIndex is the location within the calling string to start the search from. It can be any integer from zero to *stringName*.length - 1.

Characters in a string are indexed from left to right. The index of the first character is zero, and the index of the last character is *stringName*.length - 1.

If you do not specify a value for *fromIndex*, lastIndexOf assumes *stringName*.length - 1 (the end of the string) by default. If *searchValue* is not found, lastIndexOf returns -1.

Example

The following example uses indexOf and lastIndexOf to locate values in the string “Brave new world.”

```
var anyString="Brave new world"

//returns 8
anyString.indexOf("w")
//returns 10
anyString.lastIndexOf("w")
//returns 6
anyString.indexOf("new")
//returns 6
anyString.lastIndexOf("new"))
```

7.7.3.3 substring

Returns a subset of a *string* object.

stringName`.substring(`*indexA*`, `*indexB*`)`

stringName is any string.
indexA is any integer from zero to *stringName*.length - 1,.
indexB is any integer from zero to *stringName*.length - 1,.

Characters in a string are indexed from left to right. The index of the first character is zero, and the index of the last character is *stringName*.length - 1.

If indexA is less than indexB, the substring method returns the subset starting with the character at indexA and ending with the character before indexB. If indexA is greater than indexB, the substring method returns the subset starting with the character at indexB and ending with the character before indexA. If indexA is equal to indexB, the substring method returns the empty string.

Example

The following example uses substring to display characters from the string “Netscape”:

```
var anyString="Netscape"
```

```
//returns "Net"
anyString.substring(0,3)
anyString.substring(3,0)
//returns "cap"
anyString.substring(4,7)
anyString.substring(7,4)
```

7.7.3.4 charAt

Returns the character at the specified *index*.

stringName.charAt(*index*)

stringName is any string.
index is any integer from zero to *stringName*.length - 1,.

Characters in a string are indexed from left to right. The index of the first character is zero, and the index of the last character is *stringName*.length - 1. If the *index* you supply is out of range, JavaScript returns an empty string.

Example

The following example displays characters at different locations in the string “Brave new world”:

```
var anyString="Brave new world"

// The character at index 0 is B
anyString.charAt(0))
// The character at index 3 is v
anyString.charAt(3)
```

7.7.3.5 toLowerCase

Returns the calling string value converted to lowercase.

stringName.toLowerCase()

stringName is any string.

The toLowerCase method returns the value of stringName converted to lowercase. toLowerCase does not affect the value of stringName itself.

Example

The following example returns the lowercase string “alphabet”:

```
var upperText="ALPHABET"
upperText.toLowerCase()
```

7.7.3.6 toUpperCase

Returns the calling string value converted to uppercase.

stringName.toUpperCase()

stringName is any string.

The toUpperCase method returns the value of stringName converted to uppercase. toUpperCase does not affect the value of stringName itself.

Example

The following example returns the string “ALPHABET”:

```
var lowerText="alphabet"
lowerText.toUpperCase()
```

7.7.3.7 split

Splits a String object into an array of strings by separating the string into substrings. Returns an Array object. Syntax:

stringName.split(*separator*)

stringName is a String object.
separator is string literal or expression that separates the string into substrings.

Example

Example TBD.