JavaScript Language Specification

Preliminary Draft

Brendan Eich
C. Rand McKinney
Netscape Communications Corp.

JavaScript 1.1

11/18/96

Contents

Note: Page numbers are incorrect in this version.

Chapter

1

Introduction

JavaScript is a general-purpose, prototype-based, object-oriented scripting language. It is designed to be embedded in diverse applications and systems, without consuming much memory. JavaScript borrows most of its syntax from Java, but also inherits from Awk and Perl, with some indirect influence from Self in its object prototype system.

JavaScript is dynamically typed, that is, programs do not declare variable types, and the type of a variable is unrestricted and can change at runtime. Source code can be generated at runtime and evaluated against an arbitrary scope. Typical implementations compile by translating source into an unspecified bytecode format, to check syntax and source consistency. Note that the ability to generate and interpret programs at runtime implies the presence of a compiler at runtime.

JavaScript is a high-level scripting language that does not depend on or expose particular machine representations or operating system services. It provides automatic storage management, typically using a garbage collector. The language and the standard objects and functions documented in this specification provide no unsafe access to memory or other hardware resources.

1.1 Implementation versions

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 its standard objects and functions, but not the objects and functions peculiar to a particular implementation. Where possible, bugs in the implementation of version 1.1 are identified as deviations from the specification.

Although JavaScript was originally implemented as a scripting language for HTML in Netscape Navigator and LiveWire, this specification does not prescribe any particular application.

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 informal grammar in this specification is bottom-up and left-recursive. A complete top-down, right-recursive grammar with disambiguation rules is included in an appendix.

The subscripted suffix $_{opt,}$ 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:

```
OctalDigit:
    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

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

Lexical structure

This chapter defines JavaScript's lexical grammar by specifying how input characters may be composed into white space, comments, and tokens.

2.1 Character set

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

2.2 Lexical translations

The translation of an ASCII character stream into a sequence of JavaScript tokens uses 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 characters, which are the input 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 Comments

JavaScript has two kinds of comments:

A traditional C-style comment: all the text from /* to */ is ignored:

```
/* text */
```

A single-line C++-style comment: all the text from // to the end of the line is ignored:

```
// text
```

These comments are formally specified by the following lexical grammar:

```
Comment:
    TraditionalComment
    SingleLineComment

TraditionalComment:
    /* CommentText opt */

CommentText:
    CommentCharacter
    CommentText CommentCharacter

CommentCharacter:
    NotStarSlash
    / NotStar
    * NotSlash
    LineTerminator

NotStar
    InputCharacter, but not *

NotSlash
    InputCharacter, but not /

NotStarSlash
    InputCharacter, but not * and not /

SingleLineComment:
    // CharactersInLine opt 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        interface   this
               boolean     extends     long        throw
               break       final       native      throws
               byte        finally     new         transient
               case        float       package     try
               catch       for         private     typeof
               char        function    protected   var
               class       goto        public      void
               const       if          return      volatile
               continue    implements  short       while
               default     import      static      with
               delete      in          super
               do          instanceof  switch
               double      int         synchronized
```

The above list includes all keywords used currently and reserved for future use. The following table lists keywords used in JavaScript version 1.1:

```
                           break       new
                           continue    return
                           delete      this
                           else        typeof
                           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 literal.

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:
    IdentifierChars, but not a *Keyword* or *BooleanLiteral* or *NullLiteral*

```
IdentifierChars:
    JavaScriptLetter
    IdentifierChars JavaScriptLetterOrDigit

JavaScriptLetter:
    any uppercase or lowercase ASCII letter (a-z, A-Z)

    _
    $

JavaScriptLetterOrDigit:
    JavaScriptLetter
    any digit (0-9)
```

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 lone 0, or a digit from 1 to 9 followed by zero or more digits from 0 to 9, and represents a nonnegative 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 nonnegative 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 nonnegative integer.

```
OctalLiteral:
    0 OctalDigit
    OctalLiteral OctalDigit

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

The largest hexadecimal and octal literals are 0xffffffff and 037777777777, respectively, which equal 4294967295. A compile-time error occurs for any integer literal of greater value.

Examples of integer literals:

```
    0
    1996
    0372
    0xDeadBeef
    0x00FF88FF
```

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 or an exponent, are required. All other parts are optional.

```
FloatingPointLiteral:
    Digits . Digits_opt ExponentPart_opt
    . Digits ExponentPart_opt
    Digits ExponentPart

ExponentPart:
    ExponentIndicator SignedInteger

ExponentIndicator: one of
    e E

SignedInteger:
    Sign_opt Digits

Sign: one of
    + -
```

The largest positive finite floating point literal is 1.79769313486231570e+308. The smallest positive finite floating point literal is 4.94065645841246544e-324.

A compile-time error occurs if a non-zero floating point literal is too large, so that on rounded conversion to its internal representation, it becomes an IEEE 754 infinity.[1] A JavaScript program can represent infinities without producing a compile-time error by using the predefined constants Number.POSITIVE_INFINITY and Number.NEGATIVE_INFINITY.

A compile-time error occurs if a nonzero floating-point literal is too small, so that, on rounded conversion to its internal representation, it becomes a zero. A compile-time error does not occur if a nonzero floating-point literal has a small value that, on rounded conversion to its internal representation, becomes a nonzero denormalized number.[2]

Examples of floating-point literals are:

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

2.7.3 Boolean literals

The boolean type has two values, represented by the literals 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:
    " StringCharactersDQopt "
    ' StringCharactersSQopt '

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.

---

[1] . JavaScript 1.1 as implemented in Navigator 3.0 fails to report this error.

[2] . JavaScript 1.1 as implemented in Navigator 3.0 fails to report this error.

It is a compile-time error for a line terminator to appear after the opening " and before the closing ". A long string literal can be broken up into shorter pieces and written as a 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
    HexEscape

OctalEscape:
    \ OctalDigit
    \ OctalDigit OctalDigit
    \ ZeroToThree OctalDigit OctalDigit

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

ZeroToThree: one of
    0 1 2 3

HexEscape:

    \ x *HexDigit 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 is 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. Note that dot (.) is an operator in JavaScript, wheras it is a separtor in Java.

Operator: one of:

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

Chapter

3

Types, values, and variables

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, number, and undefined; its reference types are string, object (including the null object), and function. Strings compare by value (ASCII lexicographical order), not reference, when used as operands of the equality and relational operators.

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. The integer bitwise-logical and shift operators work with 32-bit signed two's-complement integers. Floating-point numbers are in 64-bit IEEE 754 format.

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.

Each type has a corresponding object class: Boolean, Number, String, Object, and Function. JavaScript converts values to objects by constructing an object of the corresponding class for the value's type.

3.1.1 Type names

The typeof operator returns a string naming the type of its operand, as described in 4.5.10 The typeof operator.

3.1.2 Type conversion

As summarized in the following table, JavaScript performs automatic type conversion at runtime. The From type of an operand is its type after evaluation. The To type is the type required by its combination with another operand and a binary operator, or by a unary operator.

		To type				
		function	**object**	**number**	**boolean**	**string**
From type	**undefined**	error	null	error	false	"undefined"
	function	N/C	Function object	valueOf/error	valueOf/true	decompile
	object					
	(not null)	Function object	N/C	valueOf/error	valueOf/true	toString/valueO
	(null)	error		0	false	f[3] "null"
	number					
	(zero)	error	Number	N/C	false	"0"
	(nonzero)	error	Number		true	default*
	(NaN)	error	Number		false[4]	"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/error	true	

Key

When two results separated by a slash are given, JavaScript tries the first, and if unsuccessful, uses the second.

N/C: No conversion necessary.
decompile: A string containing the function's canonical source.
toString: The result of calling the toString method.
valueOf: The result of calling the valueOf method, if it returns a value of the To type.
number: Numeric value if string is a valid integer or floating-point literal.

3.1.3 The toString method

Every object has a toString method used to convert the object to a string value, as follows:

- Boolean objects are converted to the string literal expression of their value: "true" or "false."
- Number objects are converted to the string literal expression of their value: decimal if integer and floating-point otherwise.
- Functions are decompiled, that is, a string containing the function's source definition is pretty-printed.

[3] . If valueOf does not return a string, the default object-to-string conversion is used.

[4]. JavaScript 1.1 as implemented in Navigator 3.0 converts NaN to true.

- Objects are converted by trying the object's toString method. If that is not defined, then the object's valeuOf method is tried. If that does not exist or does not return a string, then the result is a string of the form "[object *class*]", where class is the capitalized type name.

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
n0 = 0
n1 = 123

println(true.toString())
println(false.toString())
println(f.toString())
println(objnull)
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]
}
[object Object]
0
123
Infinity
```

3.1.4 The valueOf method

Every object has a valueOf method that returns the value associated with the object, if any. For Boolean and Number objects, valueOf returns the primitive boolean or number value passed to the object's constructor. For String and Function objects, valueOf returns the string or function reference that was passed to the constructor. If an object has no associated value, valueOf returns the object reference itself.

3.2 Primitive types and values

A primitive type is predefined by the JavaScript language:

```
PrimitiveType:
    boolean
    number
    undefined
```

3.2.1 Boolean types and values

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

3.2.2 Boolean operations

JavaScript's Boolean operators treat their operands as boolean values. The logical-not operator returns a boolean result; the remaining operators return one of their operands as their result, with no conversion of the result to boolean.

JavaScript's Boolean operations are:

- The logical-not operator !
- The logical-and operator &&. If the first operand of && converts to false, the result is the first operand's value and the second operand is not evaluated; otherwise the result is the second operand's value.
- The logical-or operator ||. If the first operand of || converts to true, the result is the first operand's value and the second operand is not evaluated; otherwise the result is the second operand's value.
- The conditional operator ?:. If the first operand (which precedes the ?) converts to true, the result is the value of the second operand (between the ? and :). Otherwise the result is the value of the third operand (which follows the :).

3.2.3 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) value. 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.

3.2.4 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 relational operators <, <=, >, and >=
  - The 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 ?:

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 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.

3.2.4.1 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.

3.2.5 The undefined type

Any variable that has not been assigned a value is of type undefined. The undefined type has one value, undefined. The void operator evaluates its operand, discards the value, and results in undefined.

3.3 Reference types and values

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

```
ReferenceType:
    String
    Object
    Function
```

3.3.1 String types and operations

A string is a sequence of ASCII characters created by use of a string literal or a string expression.[5] Every string has a length property that is an integer equal to the number of characters in the string. A string converts to a String object that has a number of built-in methods, 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 an operand of another type, it will convert the second operand to string, as described in 3.1.2 Type Conversion.
- 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.

3.3.2 Object types and operations

An object is a container for properties. A value of type object is a pointer to such a container, or a special null reference, which refers to no object. Each property of an object can be of any type and can be named by any string. A property name that is a nonnegative integer literal is called an index. A property name that is an identifier (as defined in) can be used after the dot operator (.).

The object operators are:

- The dot operator .

[5] . JavaScript 1.1 as implemented in Navigator 3.0 disallows the ASCII NUL character in strings.

- The index operator [ ]
- The new operator
- The delete operator.

3.3.2.1 The null object

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

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:
    identifier = new Function("Block")
    identifier = new Function(parameterList, "Block")

parameterList:
    "identifier"
    "identifier", parameterList
```

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

A function object is of type object not function. The only way to create a value of type function is with a function definition.

Examples

Here is a standard definition of a simple factorial function:

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

Here is the same function defined as a function object:

```
fact = new Function("n", "if (n <= 1) return 1; return n * fact(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 assignment or by the ++ (increment) or -- (decrement) operators.

3.4.1 Variable lifetime

JavaScript has four kinds of variables, distinguished by lifetime:

- Global variables name properties of the global object. They have lifetimes as long as any program execution or function definition and possibly longer, depending on the system in which JavaScript is embedded.
- Object properties name values contained by an object. They have the same lifetime as the containing object.
- Function parameters name argument values passed to a function. They have the same lifetime as the execution of a specific function call.
- Local variables name values associated with a particular function call. They have the same lifetime as the execution of a specific function call.

3.4.2 Initial values of variables

Every variable in a JavaScript program has a value:

- If a global or local variable is used before it is set by assignment, its value is undefined.
- If an object property is used before it is set by assignment, its value is undefined.[6]
- A function parameter is initialized to the corresponding argument value provided in the function call. If there is no corresponding argument, the parameter's value is undefined.

3.5 Names

A name is an identifier used to refer to an variable or a function in a JavaScript program. 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 a name is associated with an object.

3.5.1 Scope resolution

All names have scope. The name space in which functions are defined is called the *global scope*. Qualified names identify the scope explicitly as an object reference to the left of the dot operator. Simple names have implicit scope determined at run-time as follows:

[6] . In JavaScript 1.1 as implemented in Navigator 3.0, indexed properties default to null.

- Each enclosing with statement's object is searched, starting from the innermost with statement. If the simple name identifies a property of the object, the name is qualified by that object, and its scope is determined.
- The function currently executing (if any) is searched for local variables and function parameters identified by the simple name. If found, the simple name's scope is the function.
- Depending on the system in which JavaScript is embedded, other objects whose scopes enclose the function currently executing (if any) are searched, starting from the innermost object, for a property identified by the simple name. If found, the simple name's scope is the object containing the property.
- The global scope is searched for a property identified by the simple name. If found, the simple name's scope is the global scope.

3.5.2 Declaration and visibility

JavaScript variables may be implicitly declared by assignment, or explicitly declared by a var statement. For example, both of the following statements declare the variable x if it has not been declared yet:

```
x = 42
var x
```

The second statement declares x but does not initialize it. Uninitialized variables have the value undefined. A var statement can initialize the variable it declares with an expression, called an *initializer*:

```
var x = 42
```

Implicit declarations take effect at runtime, in the normal order of execution. Explicit declarations have at most two effects, one at compile-time that defines a property of the variable's enclosing scope, and (if there is an initializer) a runtime effect that initializes the variable to the value of its initializer. A variable is *visible* once its implicit declaration has been executed, or its explicit declaration has been compiled.

A variable declared outside a function is a global variable, and once visible, is accessible everywhere in the global scope. A variable implicitly declared within a function is also a global variable. A variable declared with var within a function is a local variable, and is accessible only within that function.

3.5.3 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, 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

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 is 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 calls.

An expression is void if it gets its value from a function call 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 operand of a binary operator is fully evaluated before any part of the right operand is evaluated. For example, if the left operand contains an assignment to a variable and the right 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:

```
i = 2
j = (i=3) * i
println(j)
```

prints

9

And:

```
a = 9
a += (a =3) // first example
println(a)
b = 9
b = b + (b = 3) // second example
println(b)
```

prints

12
12

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

In a function or constructor call, 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 calls, and array accesses.

```
PrimaryExpr:
    Literal
    this
```

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 call, otherwise it is the parent object of the function.

4.4 Member Expressions

A member expression is either a primary expression, a function call, or a named or indexed property expression.

```
MemberExpr:
    PrimaryExpr
    MemberExpr . Identifier
    MemberExpr [ Expression ]
    MemberExpr ( ArgumentList opt )
```

The definition of ArgumentList is repeated here for convenience:

```
ArgumentList:
    AssignmentExpression
    ArgumentList , AssignmentExpression
```

4.4.1 Property Expression Evaluation

A property expression combines a member expression and a property name. A property name is either an identifier to the right of the dot operator, or the string conversion of an expression enclosed by brackets.

The dot operator (.) joins an object reference to its left with an identifier to its right that names a property. If the left operand is not of object type, it is converted to object. The right operand must be an identifier or a compile-time error results. A property expression using the dot operator is called a dot expression.

The index operator ([ ]) joins an object reference to the left of the [separator to an expression after the [and before the] separator. If the left operand is not of object type, it is converted to object. The right operand is converted to string, according to the conversion rules described in 3.1.2 Type Conversion.[7] A property expression using the index operator is called an index expression.

[7] . Implementations can optimize index expressions where the index is a nonnegative integer to associate the integer, not its string conversion, with the property value. Such implementations must treat a string index containing a nonnegative integer literal as equivalent to the integer index. JavaScript 1.1 as implemented in Navigator 3.0 fails to equate integer strings with integer indexes.

4.4.2 Function Call Evaluation

This section describe the processing of a function call.

At run time, a function call requires the following steps:

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

4.4.2.1 Compute Target Reference

The member expression to the left of the argument list is evaluated and its result is converted to a function. If the conversion fails, then the program terminates. Otherwise, the function reference is the target to call.

4.4.2.2 Evaluate Arguments

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

4.4.2.3 Locate Function to Call

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 call 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 call. If there are more arguments in the call expression than in the definition, these arguments are assigned to elements of the function's arguments array.

4.5 Unary Expressions

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

```
UnaryExpression:
    MemberExpression
    ! UnaryExpression
    ~ UnaryExpression
    - UnaryExpression
    ++ MemberExpression
    -- MemberExpression
    MemberExpression ++
    MemberExpression --
    new Constructor
    delete MemberExpression
```

```
typeof UnaryExpression
void UnaryExpression
```

4.5.1 Logical Complement Operator !

The operand of the ! operator must be convertible to a boolean value or a run-time error occurs. The type of the result is boolean and its value is true if the operand converts to false and false if the operand converts to true.

4.5.2 Bitwise Complement Operator ~

The operand of the unary ~ operator must be a convertible to a number or a run-time error occurs. The type of the result is number, and its value is the bitwise complement of the 32-bit integer converted value of the operand.

4.5.3 Unary Minus Operator –

The operand of the unary - operator must be convertible to a number or a run-time error occurs. The type of the result is number, and its value is the arithmetic negation of the converted value of the operand.

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.

4.5.4 Prefix Increment Operator ++

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

At run time, the variable's value must be convertible to a number or an error occurs. The value 1 is added to the converted 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 sum is stored.

4.5.5 Prefix Decrement Operator --

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

At run time, the variable's value must be convertible to a number or an error occurs. The value 1 is subtracted from the converted 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 difference is stored.

4.5.6 Postfix Increment Operator ++

A member expression followed by a ++ operator is a postfix increment expression. The result of the member expression must be a variable, or a run-time error occurs. The result of the postfix increment expression is not a variable, but a value.

At run time, the variable's value must be convertible to a number or an error occurs. The value 1 is added to the converted 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 sum is stored.

4.5.7 Postfix Decrement Operator --

A member expression followed by a -- operator is a postfix decrement expression. The result of the member expression must be a variable, or a run-time error occurs. The result of the postfix decrement expression is not a variable, but a value.

At run time, the variable's value must be convertible to a number or an error occurs. The value 1 is subtracted from the converted 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 difference is stored.

4.5.8 The new operator

The new operator returns an object created with a constructor. The constructor is a function reference or function call expression where the function is named as in 3.5 Names. In addition to qualified names, this followed by a dot followed by a name can be used. The type of the result is object, and its value is a reference to the constructed object.

```
Constructor:
    this . ConstructorCall
    ConstructorCall

ConstructorCall:
    Identifier
    Identifier ( ArgumentList_opt )
    Identifier . ConstructorCall
```

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

4.5.9 The delete operator

The delete operator removes a property definition, frees the memory associated with it, and results in undefined.[8]

4.5.10 The typeof operator

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

The string returned is one of

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

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 "abcdef" returns "string"
```

4.5.11 The void operator

The void operator takes a unary expression of any type as its operand, evaluates it, and results in undefined.

4.6 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

[8] . In JavaScript 1.1, as implemented in Navigator 3.0, delete does not remove the property definition, instead it sets the property to null.

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

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

4.6.1 Multiplication Operator *

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

The result of a floating-point multiplication is governed by the rules of IEEE 754 double-precision 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.

4.6.2 Division Operator /

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

JavaScript does not perform integer division. The operands and result of all division operations are double-precision floating-point numbers. The result of division is determined by the specification 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.
- 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 zero.
- Division of a zero by a zero results in NaN; division of zero by any other finite value results in zero.
- 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.[9]
- 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 zero. The JavaScript language requires support of gradual underflow as defined by IEEE 754.

4.6.3 Remainder Operator %

The binary % operator is said to yield the remainder of its operands from an implied division; the left operand is the dividend and the right 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 Java 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 is zero.
- 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.

[9] . In JavaScript 1.1, as implemented in Navigator 3.0, division of a non-zero finite value by zero results in NaN.

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

4.7 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 using the + operator, if either of the operands is of type string, object, or function, then the other operand is converted to a string and the operation is a string concatenation operation (see 4.7.1 String Concatenation Operator +). In the case of an object operand, this conversion occurs only if the object has no valueOf method or its valueOf method returns a string.[10] Such an operand is called string-convertible.

If neither operand is string-convertible, the additive operators convert their operands to number, and result in a number.

4.7.1 String Concatenation Operator +

If only one operand expression is string-convertible, 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 that is the concatenation of the two strings. The characters taken from the left operand precede the characters taken from the right operand in the newly created string.

[10] . In JavaScript 1.1, as implemented in Navigator 3.0, the conversion from object to string occurs regardless of the existence or result type of the valueOf method.

4.7.1.1 String Conversion

Any object may be converted to type string by a call to the valueOf method, provided it returns a string. If it does not, and the object has a toString method, the toString method is called. For more information, see 3.1.2 Type Conversion.

4.7.1.2 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".

4.7.2 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.

The result of an addition is determined using the rules of IEEE 754 double-precision 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 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 operand is the minuend and the right operand is the subtrahend. It is always the case that a-b produces the same result as a+(-b).

4.8 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 operand of a shift operator is the value to be shifted; the right operand specifies the number of bits to shift.

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

The type of each of the operands of a shift operator must be convertible to number or a run-time error occurs. At run time, shift operations are performed on the 32-bit two's complement integer representation of the value of the left operand. The right operand is converted to an integer, and only the least five bits are used (this is known as implicit masking).

The value of n<<s is n left-shifted s bit positions; this is equivalent (only if overflow does not occur) 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 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). The added term (2<<~s) cancels out the propagated sign bit. Note that, because of the implicit masking of the right 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.

4.9 Relational Operators

The relational operators are syntactically left-associative (they group left-to- right), for example, a<b<c parses as (a<b)<c. If both operands are string-convertible (see 4.7 Additive Operators), both are converted to string, and the results are compared as strings. Otherwise, both operands must convert to number, 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.

4.9.1 String Comparison Operators

If both operands of a relational operator are string-convertible, they are converted to string and compared lexicographically.

4.9.2 Numerical Comparison Operators

If either operand of a relational operator is not string-convertible, then both must be convertible to number, or a run-time error occurs.

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

4.10 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.

4.10.1 Reference Equality Operators == and !=

If both operands are of type object or function, they are compared as pointers. Otherwise, if either operand is null, the other is converted to object and compared with null.

4.10.2 String Equality Operators == and !=

If both operands are string-convertible, but both are not object or function, and neither is null, then they are compared byte-by-byte for equality or inequality.

4.10.3 Numerical Equality Operators == and !=

If neither operand is object, function, or string-convertible, then both are converted to number, 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 operand is equal to the value of the right operand, and otherwise is false.
- The value produced by the != operator is true if the value of the left operand is not equal to the value of the right 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 Bitwise Logical Operators

The bitwise 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 logical operators convert their operands to number, which may result in a run-time error. If both operands convert to number, the results are converted to 32-bit integers, and the result is also an integral number.

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 Conditional-And Operator

The && operator is like & but evaluates its right operand only if the value of its left operand converts to 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 result type and value are the type and value of the left operand if it converts to false, otherwise the result type and value are those of the right operand.

At run time, the left operand is evaluated first; if its value converts to false, the right operand expression is not evaluated. If the value of the left operand converts to true, then the right expression is evaluated.

4.13 Conditional-Or Operator

The || operator is like | but evaluates its right operand only if the value of its left 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 result type and value are the type and value of the left operand if it converts to true, otherwise the result type and value are those of the right operand.

At run time, the left operand is evaluated first; if its value converts to true, the right operand expression is not evaluated. If the value of the left operand converts to false, then the right expression is evaluated.

4.14 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 ? AssignmentExpression : AssignmentExpression
```

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 convert to boolean or a run-time error occurs.

At run time, the first operand of the conditional expression is evaluated first and converted to boolean; the result is then used to choose either the second or the third operand expression:

- If first operand converts to true, then the second operand expression is chosen.
- If the first operand converts to false, then the third operand expression is chosen.

The chosen operand is then evaluated and the resulting value is the result of the conditional expression. The operand not chosen is not evaluated for that particular evaluation of the conditional expression.

4.15 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:
    Identifier
    PropertyExpression

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

The result of the first operand of an assignment operator must be a variable or a compile-time error occurs.[11]

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

4.15.1 Simple Assignment Operator =

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

4.15.2 Compound Assignment Operators

A compound assignment expression of the form E1 op= E2 is equivalent to E1 = (E1) op (E2), except that E1 is evaluated only once.

4.16 Comma Operator

The comma operator has the lowest precedence of all JavaScript operators. Comma expressions may not occur in function argument lists, but may occur anywhere else an expression may occur, including between brackets in an index expression.

```
Expression:
    AssigmentExpression
    Expression , AssignmentExpression
```

[11] . In JavaScript 1.1, as implemented in Navigator 3.0, the check for a variable on the left side of an assignment operator is done at run time.

The comma operator evaluates its left operand and then its right operand. Its result is the result of evaluating its right operand.

Chapter

5

Object model

The framework for the JavaScript object model is built around

- Constructor functions
- Prototype objects
- The new operator, by which a constructor function derives an object from a prototype and initializes it
- Built-in objects with pre-defined constructors and properties.

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 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 Call

A function call 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.

The number of arguments in an call does not have to match the number of arguments in the function definition. Each argument in the call 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 call will be undefined. If there are more arguments in the call 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 call expression is the value of the expression following the return statement that returned control to the call, 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 function that called it, if any. The caller proper non-null only within a function. The syntax is:

```
functionName.caller
```

If the function was not called from another function then caller is null. The function referred to by the caller property has a null caller property, even if the caller was itself called by another function.

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 call.

5.2 This

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

- For a function called by name (see 3.5 Names), this refers to the object whose scope is resolved as described in 3.5.1 Scope Resolution.
- For a function called by an index expression (see 4.4.1 Property Expression Evaluation), this refers to the object whose function-valued property is being indexed.
- Because eval is a method of every object, it follows that:
  - If eval is called as a method, this refers to the object to the left of the dot or bracket. For example:

    ```
    car.eval("this." + propertyName)
    ```

  - If eval is called as a function, then this refers to the enclosing scope's object. For example, the following function returns a reference to itself (as a function object):

    ```
    function f() {
        return eval("this")
    }
    ```

- In a call to a constructor function, this refers to the object created by the new operator.
- Outside of a function, this refers to the global object defined by the particular JavaScript implementation. For example, in Netscape Navigator, the global 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.

A constructor function sets properties of this to create the new object's properties.

5.3.1 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")
s2.rep = new_rep
println(s1.rep(3))
println(s2.rep(3))
```

The output of this example is:

```
aaa
repeat b 3 times.
```

5.3.2 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(ArgumentList)
```

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.display = displayCar
}
```

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

Statements

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

Some statements contain other statements as part of their structure; such other statements are substatements of the statement. We say that statement S immediately contains statement U if there is no statement T different from S and U such that S contains T and T contains U. In the same manner, some statements contain expressions as part of their structure.

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 assignment: JavaScript recognizes a new variable when it is set for the first time 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 = AssignmentExpression
```

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 variable declaration declares one variable, whose name is the Identifier that appears in the declarator. If the variable has no initializer, it is undefined.

The scope of variable declarations is described in 3.5.2 Declaration and Visibility.

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 convertible to boolean, or a run-time error occurs. The Expression is evaluated and converted to boolean:

- If the result is true, then the first Block is executed.
- If the result 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 associates the else 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 convertible to boolean, or a run-time error occurs.

A while statement is executed by first evaluating the Expression and converting it to boolean:

- If the result 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 result is false, no further action is taken and the while statement completes normally.

If the boolean conversion 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:
    Expression
    VariableDeclaration

ForUpdate:
    Expression
```

The *Expression*, if present, must be convertible to boolean, 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 an expression, it is evaluated, and its value, if any, is discarded.

- If the ForInit code is a variable declaration, it is executed as if it were a variable declaration statement appearing in a block.
- 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 converted to boolean, and there is then a choice:

- If the Expression is not present, or it is present and the boolean conversion of its result 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 is present, it is evaluated; its value, if any, is 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 boolean conversion of its result 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, it is evaluated; its value, if any, is 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, for, or for/in statement. this statement, called the break target, then immediately completes normally. If no while or for statement encloses the break statement, a compile-time error occurs. A break statement always completes abruptly.

6.4.6 The continue statement

The continue statement may occur only in an while, for, or for/in 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 iteration statement; this statement, called the continue target, then immediately ends the current iteration and begins a new one. If no iteration statement encloses the continue statement, a compile-time error occurs. A continue statement always completes abruptly.

6.4.7 The return statement

The return statement returns control to the caller of a function.

```
ReturnStatement:
    return Expression_opt ;
```

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

A return statement with an *Expression* transfers control to the caller of the function; the value of the Expression becomes the value of the function call.

6.4.8 The with statement

The with statement establishes the default object for a set of statements. See 3.5.1 Scope Resolution. Within the set of statements, any simple name (including the first part of a qualifed name) is resolved against the default object.

```
withStatement:
    with ( Expression ) Block
```

Expression is evaluated and converted to object. *Block* is then executed with the object pushed on a stack of default objects. The stack is popped after *Block* completes normally or abruptly. The with statement completes for the same reason that *Block* completes.

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 (LeftHandSide in Expression) Block
```

For each iteration of the loop, *Expression* is evaluated and converted to an object. For the first iteration of the loop, *LeftHandSide* is evaluated as in an *AssignmentExpression*, as described in 4.15.1 Simple Assignment Operator =, and is assigned the string identifier of the first property. Then, *Block* is executed. The second iteration occurs unless *Block* completed abruptly for the following reasons: break or return. The second iteration assigns the string identifier of the second property to *LeftHandSide*.

The loop continues until abrupt completion of *Block* due to break or return or until *Block* has completed with *LeftHandSide* set to the string identifier of the last property. Properties are ordered by the order in which they are set.

Example

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

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

6.4.10 Function definition statement

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

FunctionDefinition:
    function *Identifer* (*ParameterList*$_{opt}$) { *BlockStatements*$_{opt}$ }

ParameterList:
    Identifier
    ParameterList , Identifier

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

Example

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

Chapter

7

Built-in functions and objects

JavaScript 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. JavaScript also has constructors for Boolean and Number types.

7.1 Built-in functions

JavaScript has five functions built in to the language. They are eval, parseInt, parseFloat, escape, and unescape.

7.1.1 eval

Evaluates a string and returns a value.

`eval(`*`Expression`*`)`

Expression is evaluated, and if the result is not a string, the result is returned.

If the result is a string, it is taken to be a JavaScript program, and it is evaluated. If the program completes normally, eval returns the value of the last expression in it.

The scope of execution is determined as described in 5.2 This.

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
```

Identifier 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.

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 Xdate is an existing Date object, then

```
Xdate.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*$_{opt}$*, min*$_{opt}$*, sec*$_{opt}$)

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(expr)
```

expr is any numeric expression.

7.5.3.2 acos

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

`Math.acos(`*`expr`*`)`

expr 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(`*`expr`*`)`

expr 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(`*`expr`*`)`

expr 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(`*`expr`*`)`

expr is any numeric expression.

7.5.3.7 cos

Returns the cosine of a number.

`Math.cos(`*`expr`*`)`

expr 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(expr)
```

expr is any numeric expression.

7.5.3.9 log

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

```
Math.log(expr)
```

expr 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(expr1, expr2)
```

expr1 and *expr2* are any numeric arguments or the properties of existing objects.

7.5.3.11 min

Returns the lesser of two numbers. Syntax:

```
Math.min(expr1, expr2)
```

expr1 and *expr2* 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(`*`expr`*`)`

expr 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(`*`expr`*`)`

expr 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(`*`expr`*`)`

expr 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(`*`expr`*`)`

expr 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:

```
identifier = 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. If separator is the empty string, split separates each character into a substring element in the array.

Appendix

A

JavaScript LL(1) Grammar

This appendix contains the NQLL(1) grammar (Not Quite LL(1)) for JavaScript.

NOTE: This appendix is missing the algorithm for recovering from missing semicolon errors.

```
Program:
    empty
    Element Program

Element:
    function Identifier ( ParameterListOpt ) CompoundStatement
    Statement

ParameterListOpt:
    empty
    ParameterList

ParameterList:
    Identifier
    Identifier , ParameterList

CompoundStatement:
    { Statements }

Statements:
    empty
    Statement Statements

Statement:
    ;
    if Condition Statement
    if Condition Statement else Statement
    while Condition Statement
    ForParen ; ExpressionOpt ; ExpressionOpt ) Statement
    ForBegin ; ExpressionOpt ; ExpressionOpt ) Statement
    ForBegin in Expression ) Statement
    break ;
    continue ;
    with ( Expression ) Statement
    return ExpressionOpt ;
    CompoundStatement
    VariablesOrExpression ;
```

```
Condition:
    ( Expression )

ForParen:
    for (

ForBegin:
    ForParen VariablesOrExpression

VariablesOrExpression:
    var Variables
    Expression

Variables:
    Variable
    Variable , Variables

Variable:
    Identifier
    Identifier = AssignmentExpression

ExpressionOpt:
    empty
    Expression

Expression:
    AssignmentExpression
    AssignmentExpression , Expression

AssignmentExpression:
    ConditionalExpression
    ConditionalExpression AssignmentOperator AssignmentExpression

ConditionalExpression:
    OrExpression
    OrExpression ? AssignmentExpression : AssignmentExpression

OrExpression:
    AndExpression
    AndExpression || OrExpression

AndExpression:
    BitwiseOrExpression
    BitwiseOrExpression && AndExpression

BitwiseOrExpression:
    BitwiseXorExpression
    BitwiseXorExpression | BitwiseOrExpression

BitwiseXorExpression:
    BitwiseAndExpression
    BitwiseAndExpression ^ BitwiseXorExpression

BitwiseAndExpression:
    EqualityExpression
    EqualityExpression & BitwiseAndExpression

EqualityExpression:
    RelationalExpression
    RelationalExpression EqualityualityOperator EqualityExpression

RelationalExpression:
    ShiftExpression
    RelationalExpression RelationalationalOperator ShiftExpression

ShiftExpression:
    AdditiveExpression
    AdditiveExpression ShiftOperator ShiftExpression

AdditiveExpression:
    MultiplicativeExpression
```

```
    MultiplicativeExpression + AdditiveExpression
    MultiplicativeExpression - AdditiveExpression

MultiplicativeExpression:
    UnaryExpression
    UnaryExpression MultiplicativeOperator MultiplicativeExpression

UnaryExpression:
    MemberExpression
    UnaryOperator UnaryExpression
    - UnaryExpression
    IncrementOperator MemberExpression
    MemberExpression IncrementOperator
    new Constructor
    delete MemberExpression

Constructor:
    this . ConstructorCall
    ConstructorCall

ConstructorCall:
    Identifier
    Identifier ( ArgumentListOpt )
    Identifier . ConstructorCall

MemberExpression:
    PrimaryExpression
    PrimaryExpression . MemberExpression
    PrimaryExpression [ Expression ]
    PrimaryExpression ( ArgumentListOpt )

ArgumentListOpt:
    empty
    ArgumentList

ArgumentList:
    AssignmentExpression
    AssignmentExpression , ArgumentList

PrimaryExpression:
    ( Expression )
    Identifier
    IntegerLiteral
    FloatingPointLiteral
    StringLiteral
    false
    true
    null
    this
```