Unearthing the excellence in JavaScript

JavaScript: The Good Parts

O'REILLY® | YAHOO! PRESS

Douglas Crockford

JavaScript: The Good Parts

Douglas Crockford

Published by Yahoo Press

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

DEDICATION

For the Lads: Clement, Philbert, Seymore, Stern, and, lest we forget, C. Twildo.

Special Upgrade Offer

If you purchased this ebook directly from oreilly.com, you have the following benefits:

- DRM-free ebooks—use your ebooks across devices without restrictions or limitations
- Multiple formats—use on your laptop, tablet, or phone
- Lifetime access, with free updates
- Dropbox syncing—your files, anywhere

If you purchased this ebook from another retailer, you can upgrade your ebook to take advantage of a
these benefits for just $4.99. Click here to access your ebook upgrade.

Please note that upgrade offers are not available from sample content.

A Note Regarding Supplemental Files

Supplemental files and examples for this book can be found at http://examples.oreilly.com/9780596517748/. Please use a standard desktop web browser to access these files, as they may not be accessible from all ereader devices.

All code files or examples referenced in the book will be available online. For physical books that sh with an accompanying disc, whenever possible, we've posted all CD/DVD content. Note that while w provide as much of the media content as we are able via free download, we are sometimes limited by licensing restrictions. Please direct any questions or concerns to booktech@oreilly.com.

Preface

> *If we offend, it is with our good will That you should think, we come not to offend, But with good will. To show our simple skill, That is the true beginning of our end.*
>
> —William Shakespeare, *A Midsummer Night's Dream*

This is a book about the JavaScript programming language. It is intended for programmers who, by happenstance or curiosity, are venturing into JavaScript for the first time. It is also intended for programmers who have been working with JavaScript at a novice level and are now ready for a more sophisticated relationship with the language. JavaScript is a surprisingly powerful language. Its unconventionality presents some challenges, but being a small language, it is easily mastered.

My goal here is to help you to learn to think in JavaScript. I will show you the components of the language and start you on the process of discovering the ways those components can be put together. This is not a reference book. It is not exhaustive about the language and its quirks. It doesn't contain everything you'll ever need to know. That stuff you can easily find online. Instead, this book just contains the things that are really important.

This is not a book for beginners. Someday I hope to write a *JavaScript: The First Parts* book, but this is not that book. This is not a book about Ajax or web programming. The focus is exclusively on JavaScript, which is just one of the languages the web developer must master.

This is not a book for dummies. This book is small, but it is dense. There is a lot of material packed into it. Don't be discouraged if it takes multiple readings to get it. Your efforts will be rewarded.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, filenames, and file extensions.

`Constant width`

Indicates computer coding in a broad sense. This includes commands, options, variables, attributes, keys, requests, functions, methods, types, classes, modules, properties, parameters, values, objects, events, event handlers, XML and XHTML tags, macros, and keywords.

`Constant width bold`

Indicates commands or other text that should be typed literally by the user.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*JavaScript: The Good Parts* by Douglas Crockford. Copyright 2008 Yahoo! Inc., 978-0-596-51774-8."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at permissions@oreilly.com.

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at http://safari.oreilly.com.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596517748/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

Acknowledgments

I want to thank the reviewers who pointed out my many egregious errors. There are few things better in life than having really smart people point out your blunders. It is even better when they do it befor you go public. Thank you, Steve Souders, Bill Scott, Julien Lecomte, Stoyan Stefanov, Eric Miraglia, and Elliotte Rusty Harold.

I want to thank the people I worked with at Electric Communities and State Software who helped me discover that deep down there was goodness in this language, especially Chip Morningstar, Randy Farmer, John La, Mark Miller, Scott Shattuck, and Bill Edney.

I want to thank Yahoo! Inc. for giving me time to work on this project and for being such a great plac to work, and thanks to all members of the Ajax Strike Force, past and present. I also want to thank O'Reilly Media, Inc., particularly Mary Treseler, Simon St.Laurent, and Sumita Mukherji for making things go so smoothly.

Special thanks to Professor Lisa Drake for all those things she does. And I want to thank the guys in ECMA TC39 who are struggling to make ECMAScript a better language.

Finally, thanks to Brendan Eich, the world's most misunderstood programming language designer, without whom this book would not have been necessary.

Chapter 1. Good Parts

...setting the attractions of my good parts aside I have no other charms.

—William Shakespeare, *The Merry Wives of Windsor*

When I was a young journeyman programmer, I would learn about every feature of the languages I was using, and I would attempt to use all of those features when I wrote. I suppose it was a way of showing off, and I suppose it worked because I was the guy you went to if you wanted to know how t use a particular feature.

Eventually I figured out that some of those features were more trouble than they were worth. Some o them were poorly specified, and so were more likely to cause portability problems. Some resulted in code that was difficult to read or modify. Some induced me to write in a manner that was too tricky and error-prone. And some of those features were design errors. Sometimes language designers make mistakes.

Most programming languages contain good parts and bad parts. I discovered that I could be a better programmer by using only the good parts and avoiding the bad parts. After all, how can you build something good out of bad parts?

It is rarely possible for standards committees to remove imperfections from a language because doin so would cause the breakage of all of the bad programs that depend on those bad parts. They are usually powerless to do anything except heap more features on top of the existing pile of imperfections. And the new features do not always interact harmoniously, thus producing more bad parts.

But *you* have the power to define your own subset. You can write better programs by relying exclusively on the good parts.

JavaScript is a language with more than its share of bad parts. It went from non-existence to global adoption in an alarmingly short period of time. It never had an interval in the lab when it could be tried out and polished. It went straight into Netscape Navigator 2 just as it was, and it was very rough When Java™ applets failed, JavaScript became the “Language of the Web” by default. JavaScript’s popularity is almost completely independent of its qualities as a programming language.

Fortunately, JavaScript has some extraordinarily good parts. In JavaScript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders. The best nature of JavaScript is so effectively hidden that for many years the prevailing opinion of JavaScript was that it was an unsightly, incompetent toy. My intention here is to expose th goodness in JavaScript, an outstanding, dynamic programming language. JavaScript is a block of marble, and I chip away the features that are not beautiful until the language’s true nature reveals itself. I believe that the elegant subset I carved out is vastly superior to the language as a whole, bein more reliable, readable, and maintainable.

This book will not attempt to fully describe the language. Instead, it will focus on the good parts with

occasional warnings to avoid the bad. The subset that will be described here can be used to construct reliable, readable programs small and large. By focusing on just the good parts, we can reduce learning time, increase robustness, and save some trees.

Perhaps the greatest benefit of studying the good parts is that you can avoid the need to unlearn the bad parts. Unlearning bad patterns is very difficult. It is a painful task that most of us face with extreme reluctance. Sometimes languages are subsetted to make them work better for students. But i this case, I am subsetting JavaScript to make it work better for professionals.

Why JavaScript?

JavaScript is an important language because it is the language of the web browser. Its association wit the browser makes it one of the most popular programming languages in the world. At the same time it is one of the most despised programming languages in the world. The API of the browser, the Document Object Model (DOM) is quite awful, and JavaScript is unfairly blamed. The DOM would l painful to work with in any language. The DOM is poorly specified and inconsistently implemented. This book touches only very lightly on the DOM. I think writing a *Good Parts* book about the DOM would be extremely challenging.

JavaScript is most despised because it isn't SOME OTHER LANGUAGE. If you are good in SOME OTHER LANGUAGE and you have to program in an environment that only supports JavaScript, ther you are forced to use JavaScript, and that is annoying. Most people in that situation don't even bothe to learn JavaScript first, and then they are surprised when JavaScript turns out to have significant differences from the SOME OTHER LANGUAGE they would rather be using, and that those differences matter.

The amazing thing about JavaScript is that it is possible to get work done with it without knowing much about the language, or even knowing much about programming. It is a language with enormous expressive power. It is even better when you know what you're doing. Programming is difficult business. It should never be undertaken in ignorance.

Analyzing JavaScript

JavaScript is built on some very good ideas and a few very bad ones.

The very good ideas include functions, loose typing, dynamic objects, and an expressive object litera notation. The bad ideas include a programming model based on global variables.

JavaScript's functions are first class objects with (mostly) lexical scoping. JavaScript is the first lambda language to go mainstream. Deep down, JavaScript has more in common with Lisp and Scheme than with Java. It is Lisp in C's clothing. This makes JavaScript a remarkably powerful language.

The fashion in most programming languages today demands strong typing. The theory is that strong typing allows a compiler to detect a large class of errors at compile time. The sooner we can detect and repair errors, the less they cost us. JavaScript is a loosely typed language, so JavaScript compiler are unable to detect type errors. This can be alarming to people who are coming to JavaScript from

strongly typed languages. But it turns out that strong typing does not eliminate the need for careful testing. And I have found in my work that the sorts of errors that strong type checking finds are not th errors I worry about. On the other hand, I find loose typing to be liberating. I don't need to form complex class hierarchies. And I never have to cast or wrestle with the type system to get the behavio that I want.

JavaScript has a very powerful object literal notation. Objects can be created simply by listing their components. This notation was the inspiration for JSON, the popular data interchange format. (There will be more about JSON in Appendix E.)

A controversial feature in JavaScript is prototypal inheritance. JavaScript has a class-free object system in which objects inherit properties directly from other objects. This is really powerful, but it i unfamiliar to classically trained programmers. If you attempt to apply classical design patterns directly to JavaScript, you will be frustrated. But if you learn to work with JavaScript's prototypal nature, your efforts will be rewarded.

JavaScript is much maligned for its choice of key ideas. For the most part, though, those choices wer good, if unusual. But there was one choice that was particularly bad: JavaScript depends on global variables for linkage. All of the top-level variables of all compilation units are tossed together in a common namespace called *the global object*. This is a bad thing because global variables are evil, an in JavaScript they are fundamental. Fortunately, as we will see, JavaScript also gives us the tools to mitigate this problem.

In a few cases, we can't ignore the bad parts. There are some unavoidable awful parts, which will be called out as they occur. They will also be summarized in Appendix A. But we will succeed in avoiding most of the bad parts in this book, summarizing much of what was left out in Appendix B. I you want to learn more about the bad parts and how to use them badly, consult any other JavaScript book.

The standard that defines JavaScript (aka JScript) is the third edition of *The ECMAScript Programming Language*, which is available from http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf. The language described in this book is a proper subset of ECMAScript. This book does not describe the whole language because it leaves out the bad parts. The treatment here is not exhaustive. It avoids the edge cases. You should, too. There i danger and misery at the edges.

Appendix C describes a programming tool called JSLint, a JavaScript parser that can analyze a JavaScript program and report on the bad parts that it contains. JSLint provides a degree of rigor that is generally lacking in JavaScript development. It can give you confidence that your programs contai only the good parts.

JavaScript is a language of many contrasts. It contains many errors and sharp edges, so you might wonder, "Why should I use JavaScript?" There are two answers. The first is that you don't have a choice. The Web has become an important platform for application development, and JavaScript is th only language that is found in all browsers. It is unfortunate that Java failed in that environment; if it hadn't, there could be a choice for people desiring a strongly typed classical language. But Java did fail and JavaScript is flourishing, so there is evidence that JavaScript did something right.

The other answer is that, despite its deficiencies, *JavaScript is really good*. It is lightweight and expressive. And once you get the hang of it, functional programming is a lot of fun.

But in order to use the language well, you must be well informed about its limitations. I will pound o those with some brutality. Don't let that discourage you. The good parts are good enough to compensate for the bad parts.

A Simple Testing Ground

If you have a web browser and any text editor, you have everything you need to run JavaScript programs. First, make an HTML file with a name like *program.html*:

```
<html><body><pre><script src="program.js">
</script></pre></body></html>
```

Then, make a file in the same directory with a name like *program.js*:

```
document.writeln('Hello, world!');
```

Next, open your HTML file in your browser to see the result. Throughout the book, a `method` method is used to define new methods. This is its definition:

```
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
```

It will be explained in Chapter 4.

Chapter 2. Grammar

I know it well: I read it in the grammar long ago.

—William Shakespeare, *The Tragedy of Titus Andronicus*

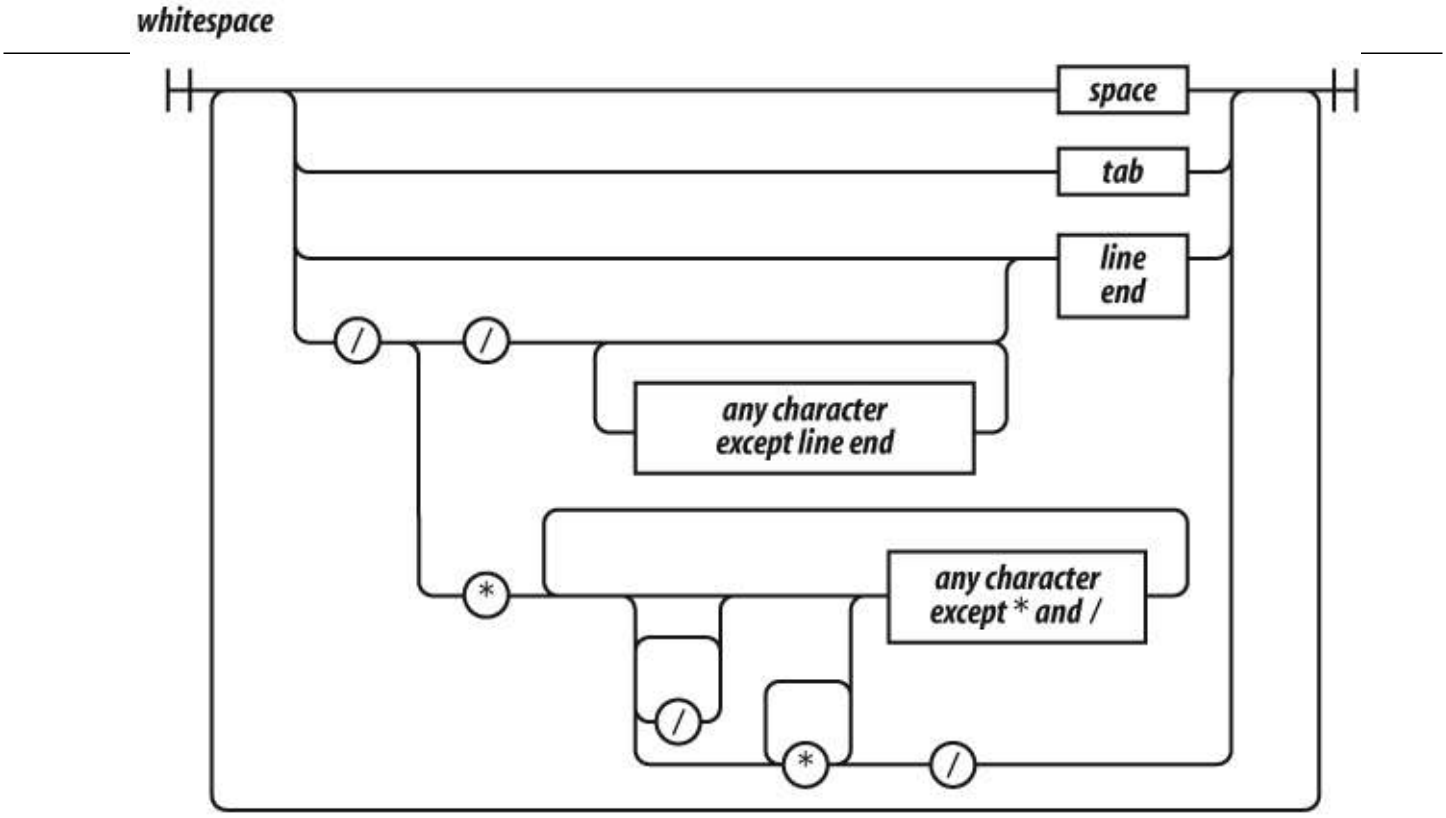
This chapter introduces the grammar of the good parts of JavaScript, presenting a quick overview of how the language is structured. We will represent the grammar with railroad diagrams.

The rules for interpreting these diagrams are simple:

- You start on the left edge and follow the tracks to the right edge.
- As you go, you will encounter literals in ovals, and rules or descriptions in rectangles.
- Any sequence that can be made by following the tracks is legal.
- Any sequence that cannot be made by following the tracks is not legal.
- Railroad diagrams with one bar at each end allow whitespace to be inserted between any pair of tokens. Railroad diagrams with two bars at each end do not.

The grammar of the good parts presented in this chapter is significantly simpler than the grammar of the whole language.

Whitespace



Whitespace can take the form of formatting characters or comments. Whitespace is usually insignificant, but it is occasionally necessary to use whitespace to separate sequences of characters that would otherwise be combined into a single token. For example, in:

```
var that = this;
```

the space between `var` and `that` cannot be removed, but the other spaces can be removed.

JavaScript offers two forms of comments, block comments formed with `/* */` and line-ending comments starting with `//`. Comments should be used liberally to improve the readability of your programs. Take care that the comments always accurately describe the code. Obsolete comments are worse than no comments.

The `/* */` form of block comments came from a language called PL/I. PL/I chose those strange pairs as the symbols for comments because they were unlikely to occur in that language's programs, except perhaps in string literals. In JavaScript, those pairs can also occur in regular expression literals, so block comments are not safe for commenting out blocks of code. For example:

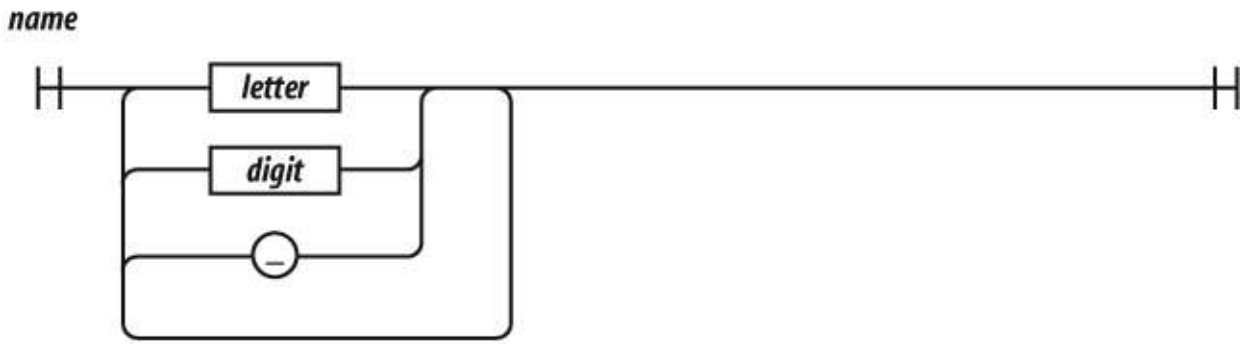
```
/*
    var rm_a = /a*/.match(s);
*/
```

causes a syntax error. So, it is recommended that `/* */` comments be avoided and `//` comments be used instead. In this book, `//` will be used exclusively.

Names

A name is a letter optionally followed by one or more letters, digits, or underbars. A name cannot be one of these reserved words:

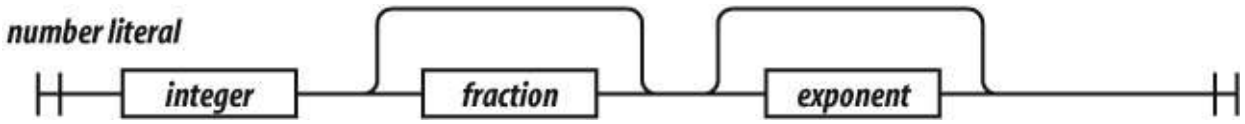
```
abstract
boolean break byte
case catch char class const continue
debugger default delete do double
else enum export extends
false final finally float for function
goto
if implements import in instanceof int interface
long
native new null
package private protected public
return
short static super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



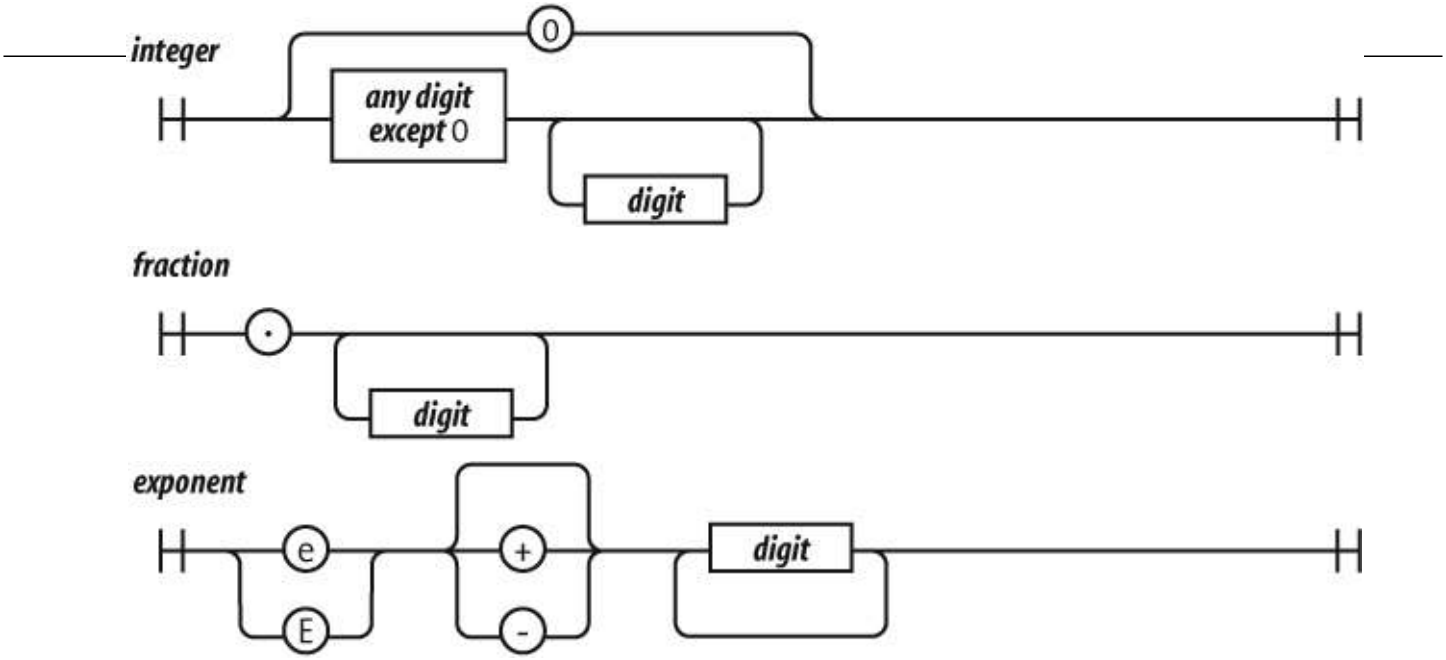
Most of the reserved words in this list are not used in the language. The list does not include some words that should have been reserved but were not, such as `undefined`, `NaN`, and `Infinity`. It is not permitted to name a variable or parameter with a reserved word. Worse, it is not permitted to use a reserved word as the name of an object property in an object literal or following a dot in a refinemen

Names are used for statements, variables, parameters, property names, operators, and labels.

Numbers



JavaScript has a single number type. Internally, it is represented as 64-bit floating point, the same as Java's `double`. Unlike most other programming languages, there is no separate integer type, so `1` and `1.0` are the same value. This is a significant convenience because problems of overflow in short integers are completely avoided, and all you need to know about a number is that it is a number. A large class of numeric type errors is avoided.



If a number literal has an exponent part, then the value of the literal is computed by multiplying the part before the `e` by `10` raised to the power of the part after the `e`. So `100` and `1e2` are the same numbe

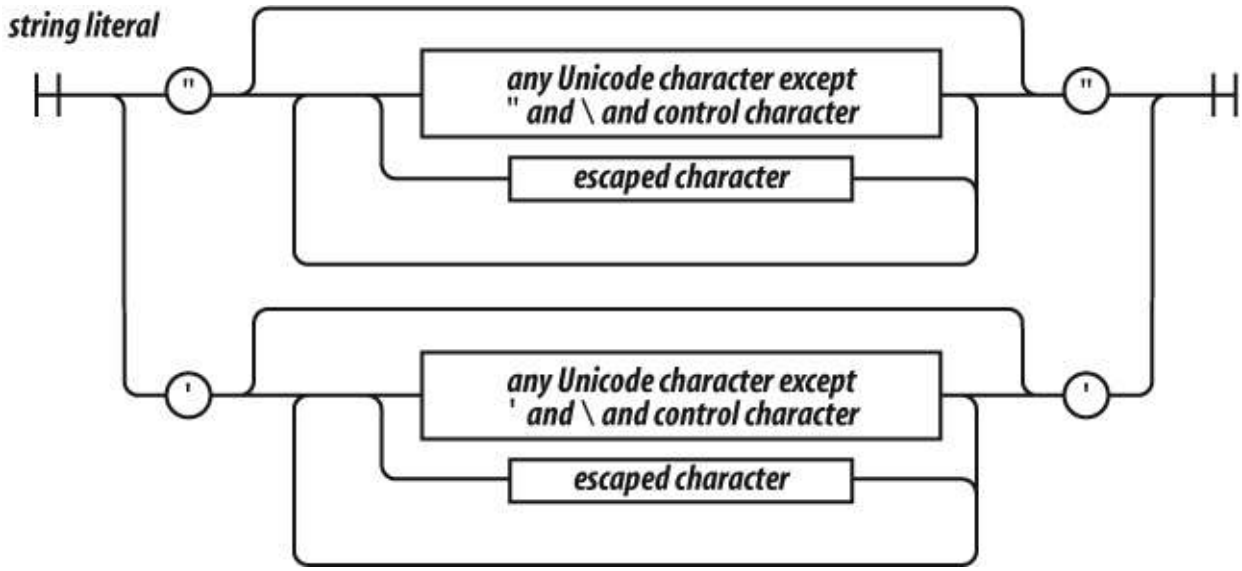
Negative numbers can be formed by using the `-` prefix operator.

The value `NaN` is a number value that is the result of an operation that cannot produce a normal result `NaN` is not equal to any value, including itself. You can detect `NaN` with the `isNaN(` *`number`* `)` function

The value `Infinity` represents all values greater than `1.79769313486231570e+308`.

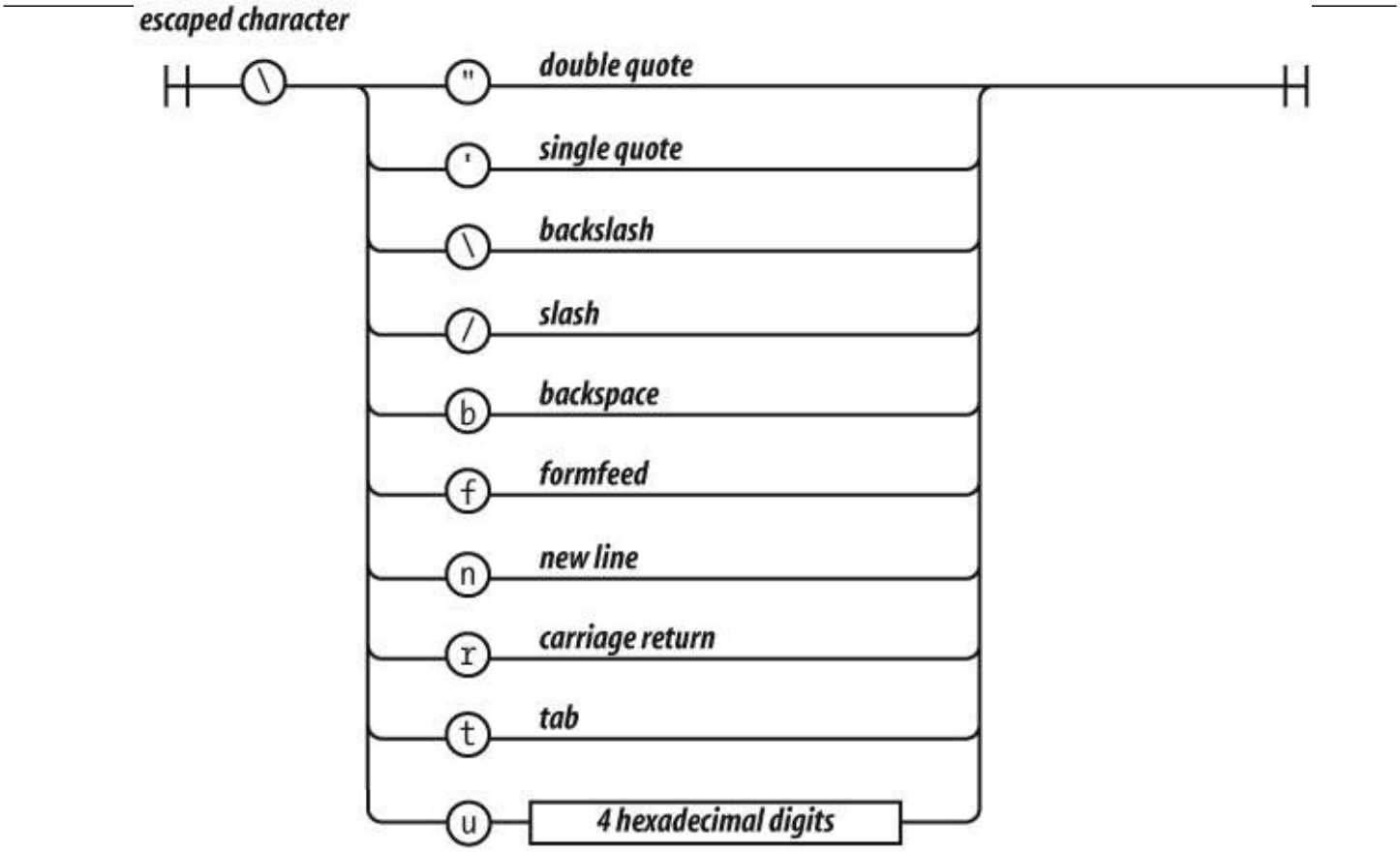
Numbers have methods (see Chapter 8). JavaScript has a `Math` object that contains a set of methods that act on numbers. For example, the `Math.floor(` *`number`* `)` method can be used to convert a numb into an integer.

Strings



A string literal can be wrapped in single quotes or double quotes. It can contain zero or more characters. The `\` (backslash) is the escape character. JavaScript was built at a time when Unicode wa

a 16-bit character set, so all characters in JavaScript are 16 bits wide.



JavaScript does not have a character type. To represent a character, make a string with just one character in it.

The escape sequences allow for inserting characters into strings that are not normally permitted, such as backslashes, quotes, and control characters. The `\u` convention allows for specifying character cod points numerically.

```
"A" === "\u0041"
```

Strings have a `length` property. For example, `"seven".length` is 5.

Strings are immutable. Once it is made, a string can never be changed. But it is easy to make a new string by concatenating other strings together with the + operator. Two strings containing exactly the same characters in the same order are considered to be the same string. So:

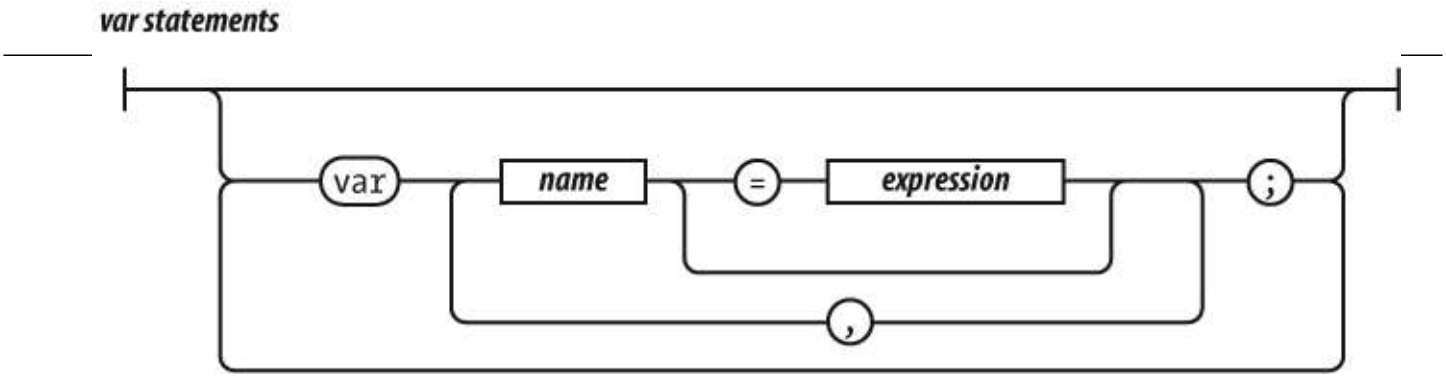
```
'c' + 'a' + 't' === 'cat'
```

is true.

Strings have methods (see Chapter 8):

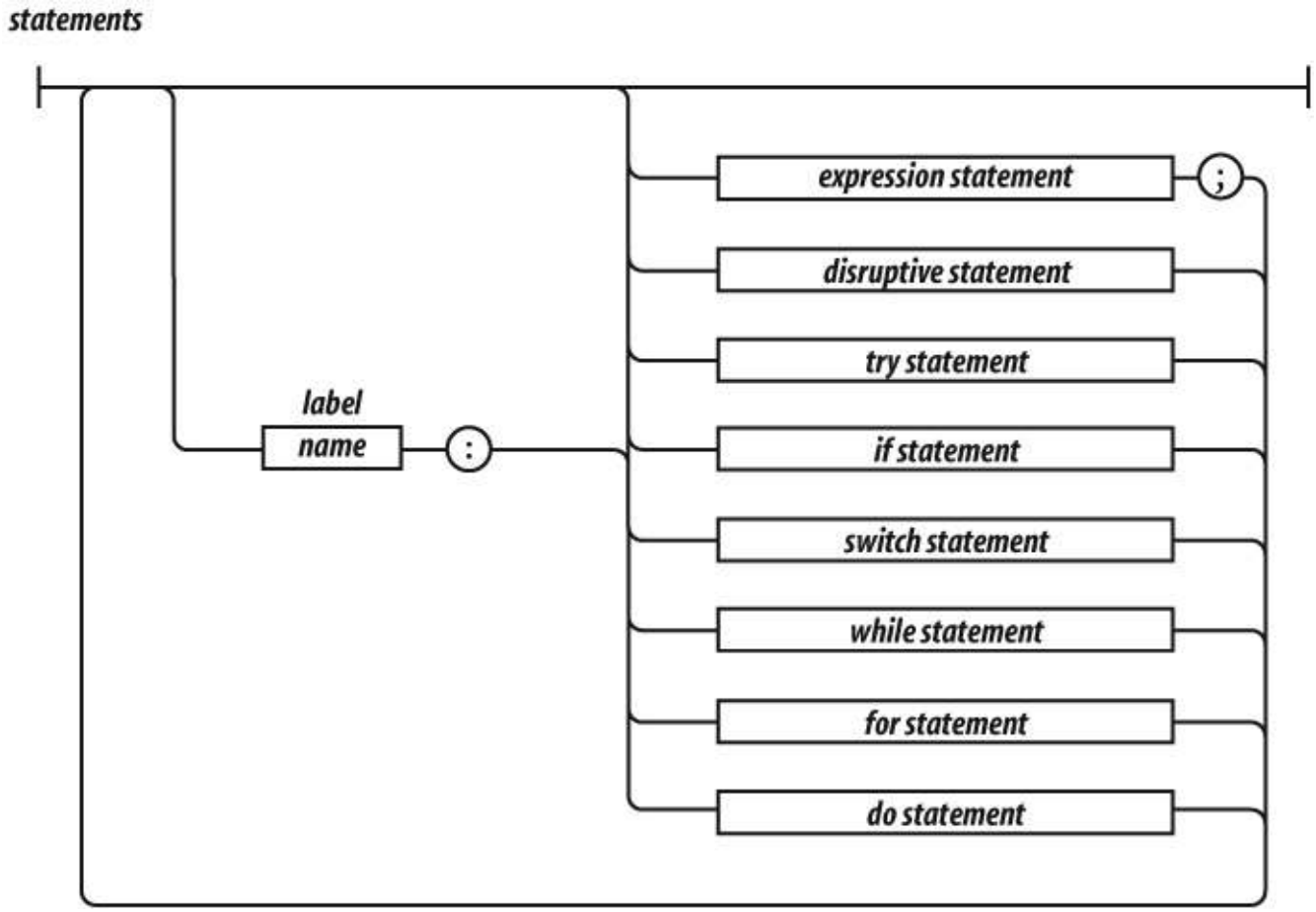
```
'cat'.toUpperCase(  ) === 'CAT'
```

Statements

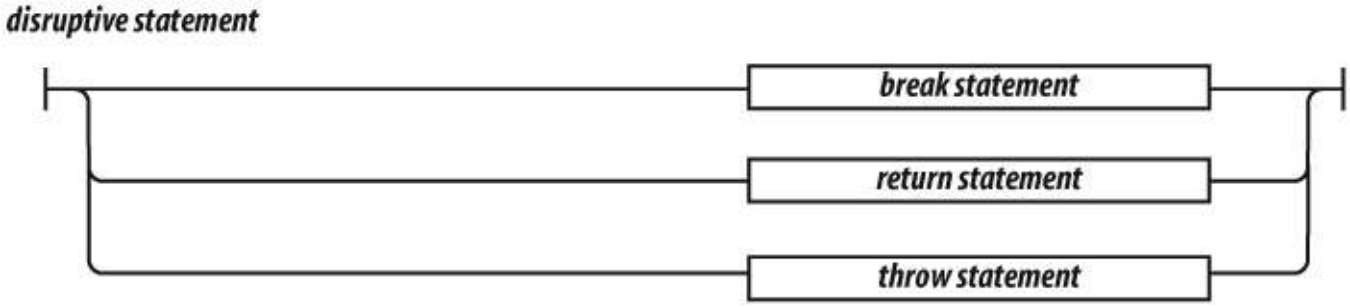


A compilation unit contains a set of executable statements. In web browsers, each `<script>` tag delivers a compilation unit that is compiled and immediately executed. Lacking a linker, JavaScript throws them all together in a common global namespace. There is more on global variables in Appendix A.

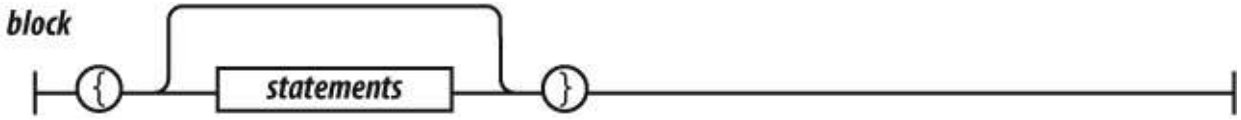
When used inside of a function, the `var` statement defines the function's private variables.



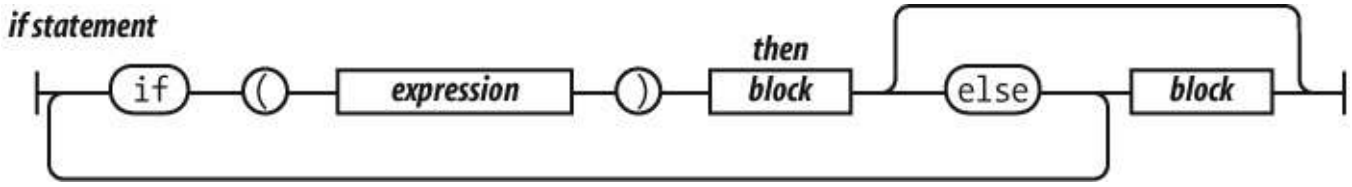
The `switch`, `while`, `for`, and `do` statements are allowed to have an optional *`label`* prefix that interact with the `break` statement.



Statements tend to be executed in order from top to bottom. The sequence of execution can be altered by the conditional statements (`if` and `switch`), by the looping statements (`while`, `for`, and `do`), by th disruptive statements (`break`, `return`, and `throw`), and by function invocation.



A block is a set of statements wrapped in curly braces. Unlike many other languages, blocks in JavaScript do not create a new scope, so variables should be defined at the top of the function, not in blocks.

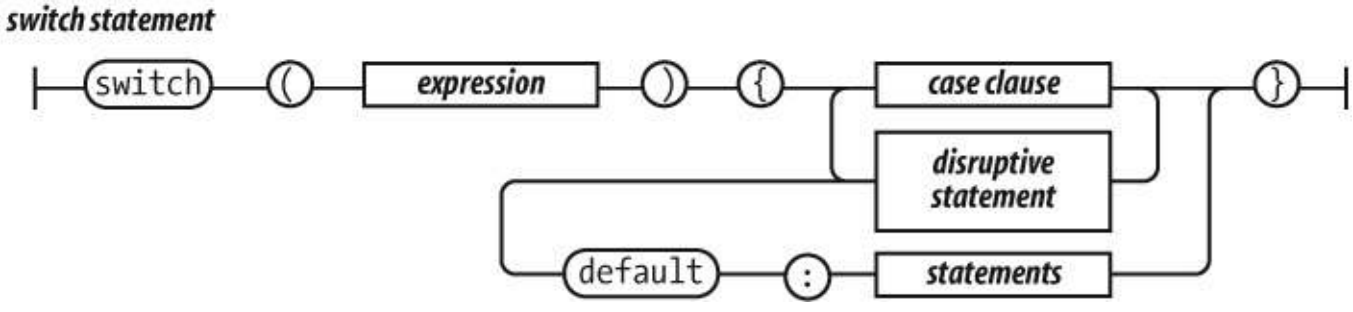


The `if` statement changes the flow of the program based on the value of the expression. The *then* block is executed if the expression is *truthy*; otherwise, the optional `else` branch is taken.

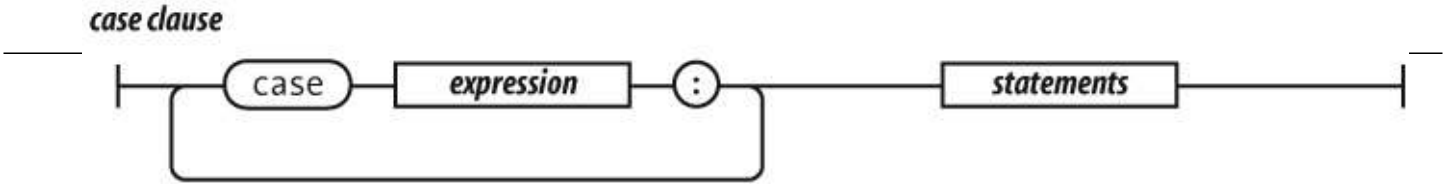
Here are the *falsy* values:

- `false`
- `null`
- `undefined`
- The empty string `''`
- The number `0`
- The number `NaN`

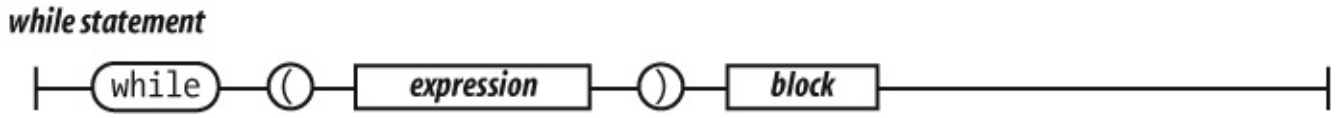
All other values are truthy, including `true`, the string `'false'`, and all objects.



The `switch` statement performs a multiway branch. It compares the expression for equality with all o the specified cases. The expression can produce a number or a string. When an exact match is found, the statements of the matching case clause are executed. If there is no match, the optional `default` statements are executed.

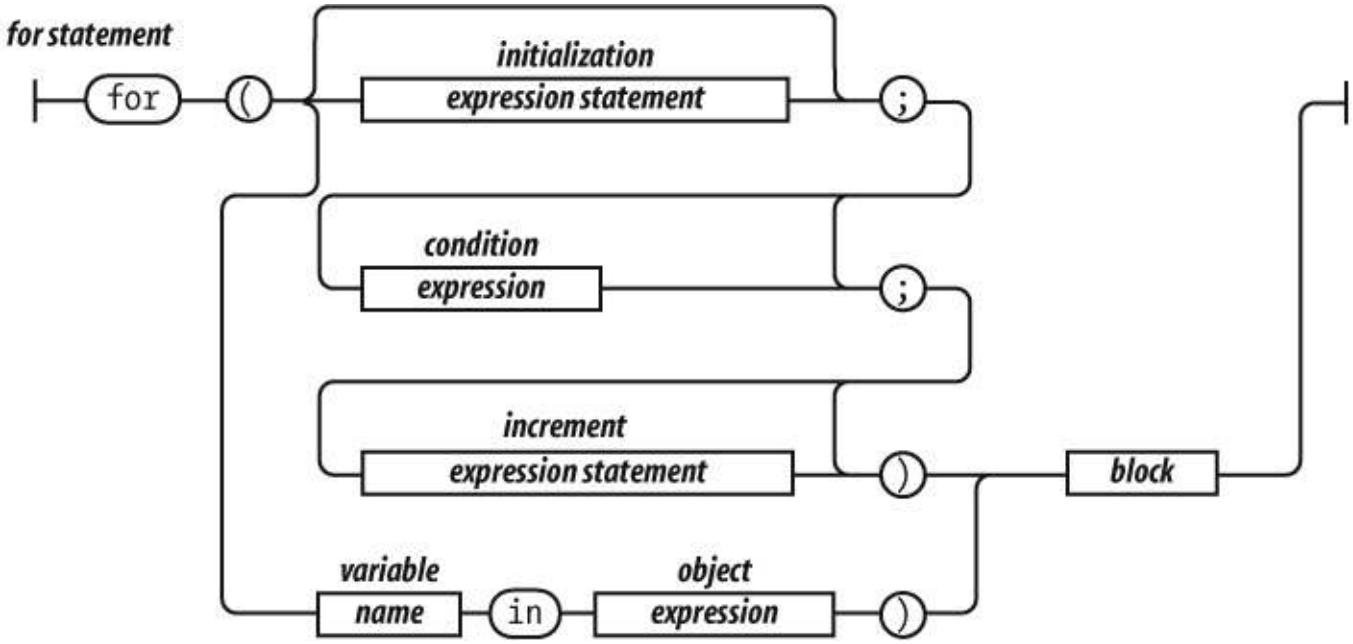


A `case` clause contains one or more case expressions. The case expressions need not be constants. Th statement following a clause should be a disruptive statement to prevent fall through into the next `case`. The `break` statement can be used to exit from a switch.



The `while` statement performs a simple loop. If the expression is falsy, then the loop will break. While the expression is truthy, the block will be executed.

The `for` statement is a more complicated looping statement. It comes in two forms.



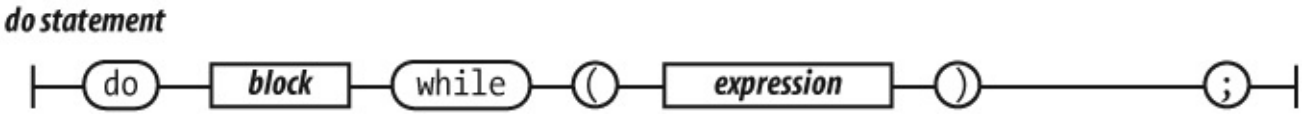
The conventional form is controlled by three optional clauses: the *initialization*, the *condition*, and th *increment*. First, the initialization is done, which typically initializes the loop variable. Then, the *condition* is evaluated. Typically, this tests the loop variable against a completion criterion. If the *condition* is omitted, then a *condition* of `true` is assumed. If the *condition* is falsy, the loop breaks. Otherwise, the block is executed, then the *increment* executes, and then the loop repeats with the *condition*.

The other form (called `for in`) enumerates the property names (or keys) of an object. On each iteration, another property name string from the *object* is assigned to the *variable*.

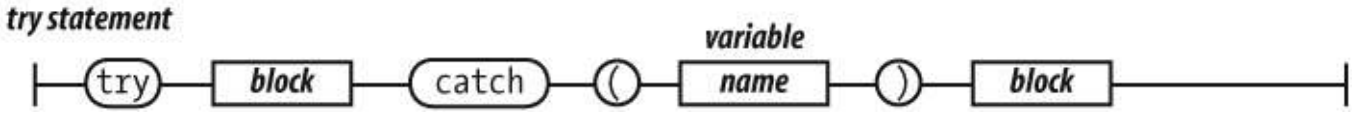
It is usually necessary to test *`object`*`.hasOwnProperty(`*`variable`*`)` to determine whether the property name is truly a member of the object or was found instead on the prototype chain.

```
for (myvar in obj) {
    if (obj.hasOwnProperty(myvar)) {
        ...
    }
```

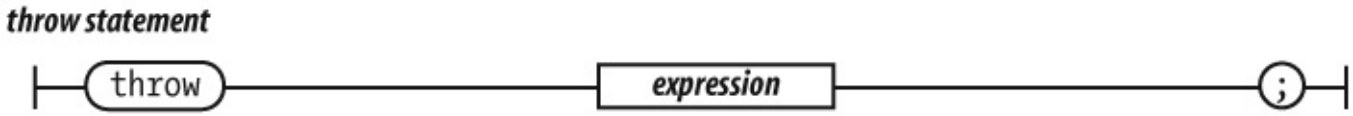
```
}
```



The `do` statement is like the `while` statement except that the expression is tested after the block is executed instead of before. That means that the block will always be executed at least once.

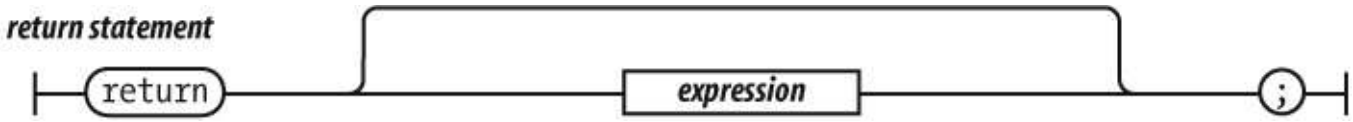


The `try` statement executes a block and catches any exceptions that were thrown by the block. The `catch` clause defines a new *variable* that will receive the exception object.



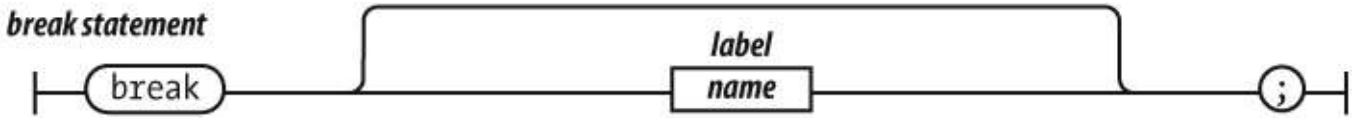
The `throw` statement raises an exception. If the `throw` statement is in a `try` block, then control goes t the `catch` clause. Otherwise, the function invocation is abandoned, and control goes to the `catch` clause of the `try` in the calling function.

The expression is usually an object literal containing a `name` property and a `message` property. The catcher of the exception can use that information to determine what to do.



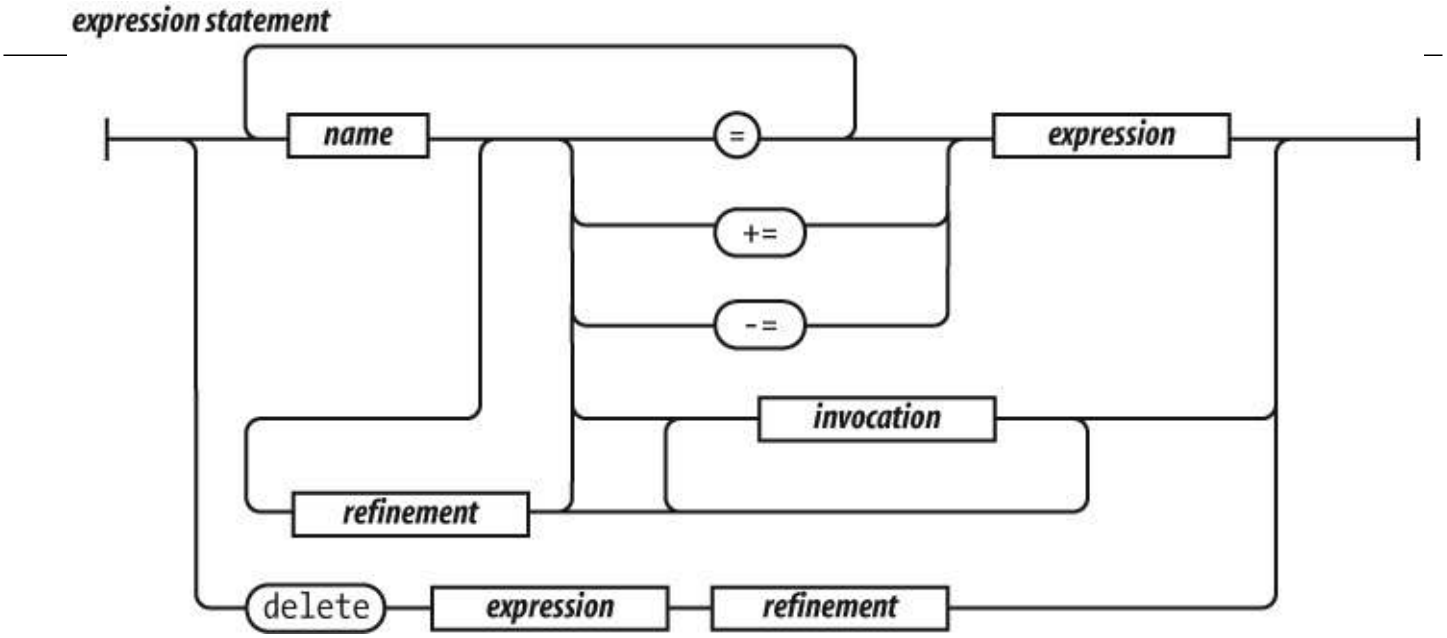
The `return` statement causes the early return from a function. It can also specify the value to be returned. If a return expression is not specified, then the return value will be `undefined`.

JavaScript does not allow a line end between the return and the expression.



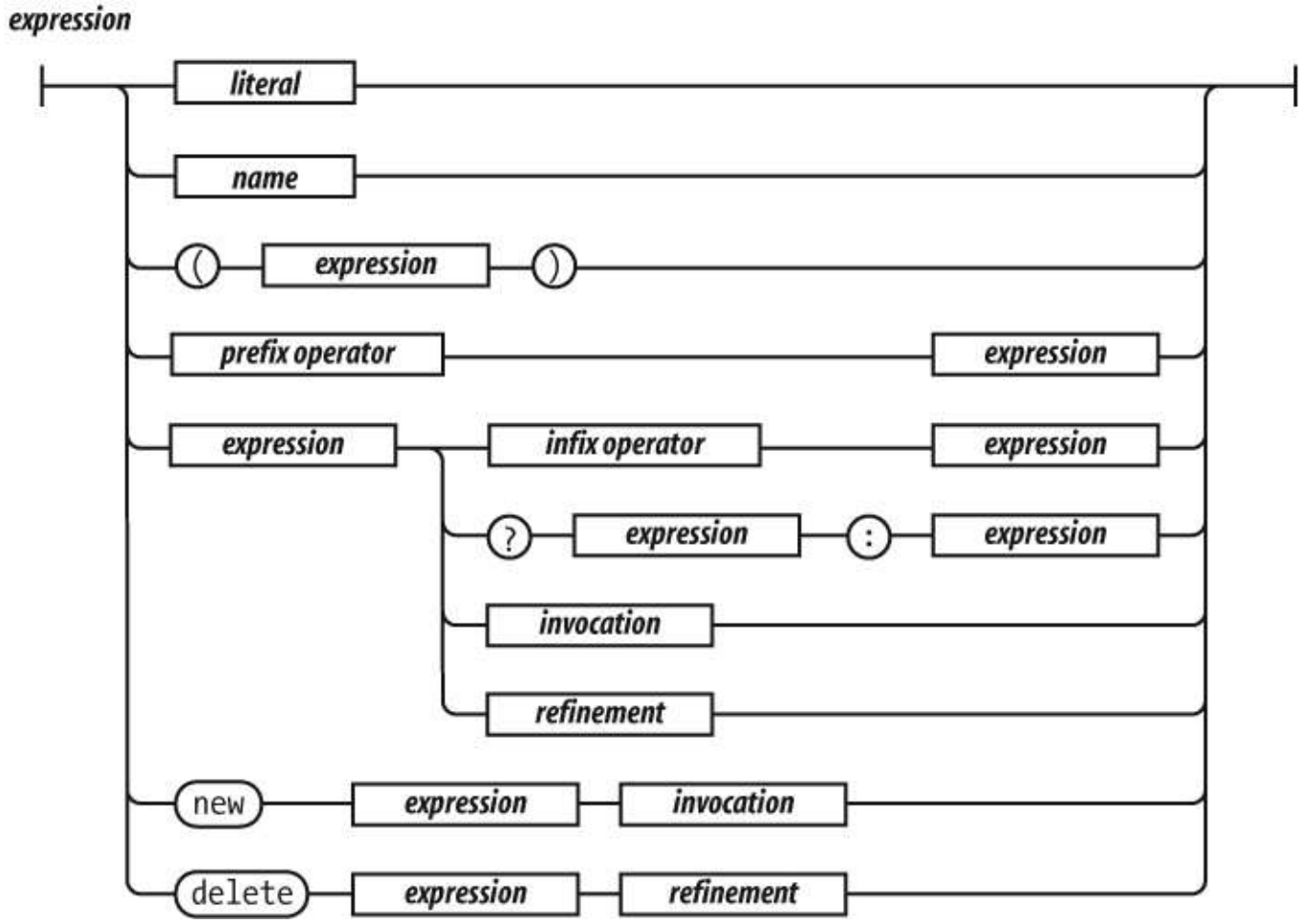
The `break` statement causes the exit from a loop statement or a `switch` statement. It can optionally have a *label* that will cause an exit from the labeled statement.

JavaScript does not allow a line end between the break and the label.



An `expression` statement can either assign values to one or more variables or members, invoke a method, delete a property from an object. The = operator is used for assignment. Do not confuse it with the === equality operator. The += operator can add or concatenate.

Expressions



The simplest expressions are a literal value (such as a string or number), a variable, a built-in value (`true`, `false`, `null`, `undefined`, `NaN`, or `Infinity`), an invocation expression preceded by `new`, a

refinement expression preceded by `delete`, an expression wrapped in parentheses, an expression preceded by a prefix operator, or an expression followed by:

- An infix operator and another expression
- The `?` ternary operator followed by another expression, then by `:`, and then by yet another expression
- An invocation
- A refinement

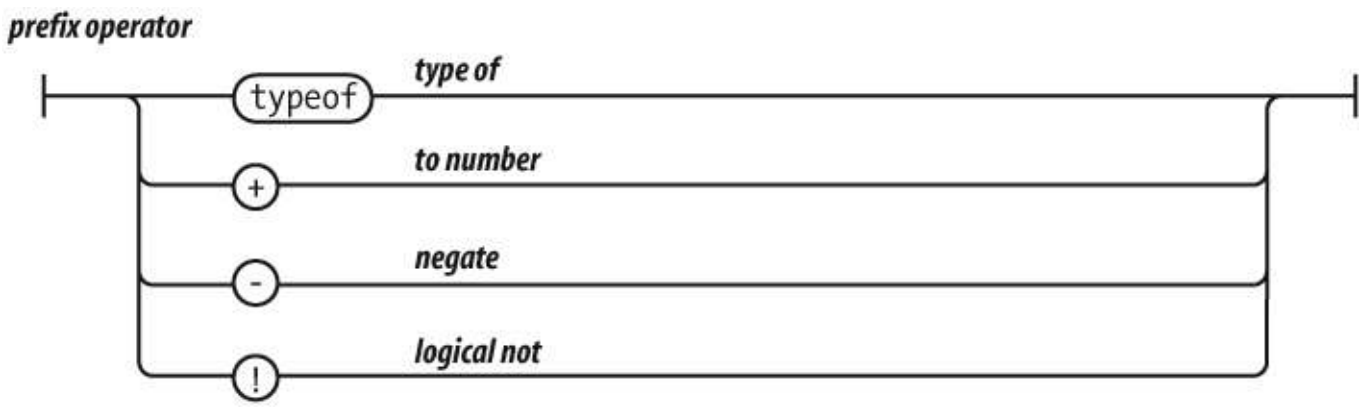
The `?` ternary operator takes three operands. If the first operand is truthy, it produces the value of the second operand. But if the first operand is falsy, it produces the value of the third operand.

The operators at the top of the operator precedence list in Table 2-1 have higher precedence. They bind the tightest. The operators at the bottom have the lowest precedence. Parentheses can be used to alter the normal precedence, so:

```
2 + 3 * 5 === 17
(2 + 3) * 5 === 25
```

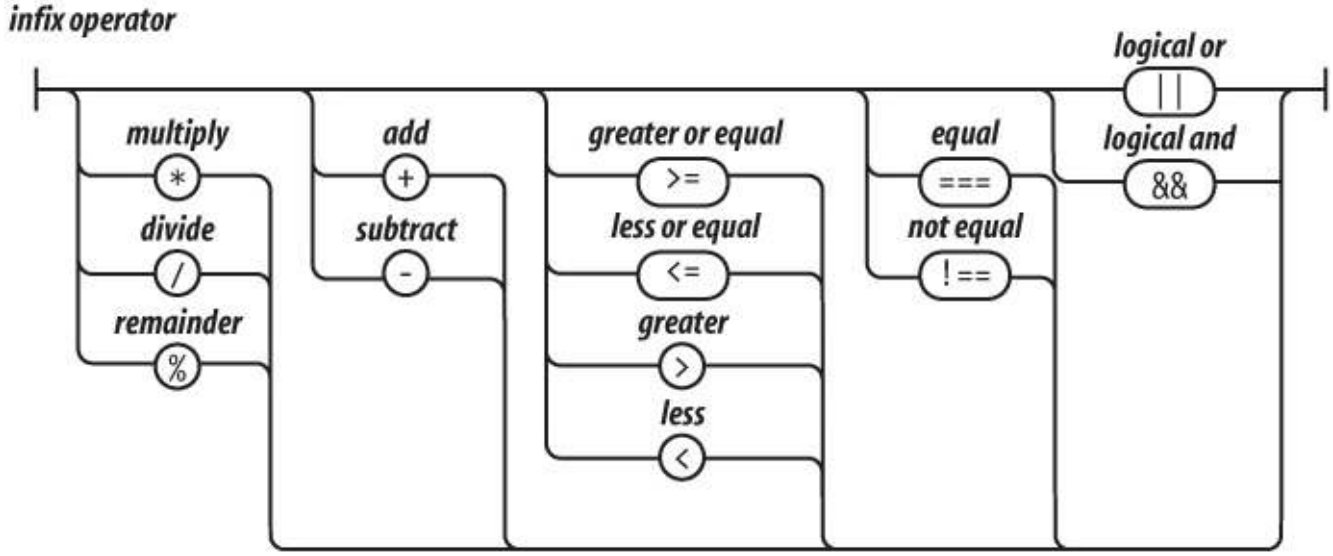
Table 2-1. Operator precedence

`. [] ( )`	Refinement and invocation
`delete new typeof + - !`	Unary operators
`* / %`	Multiplication, division, remainder
`+ -`	Addition/concatenation, subtraction
`>= <= > <`	Inequality
`=== !==`	Equality
`&&`	Logical and
`||`	Logical or
`?:`	Ternary



The values produced by `typeof` are `'number'`, `'string'`, `'boolean'`, `'undefined'`, `'function'`, and `'object'`. If the operand is an array or `null`, then the result is `'object'`, which is wrong. There will be more about `typeof` in Chapter 6 and Appendix A.

If the operand of `!` is truthy, it produces `false`. Otherwise, it produces `true`.

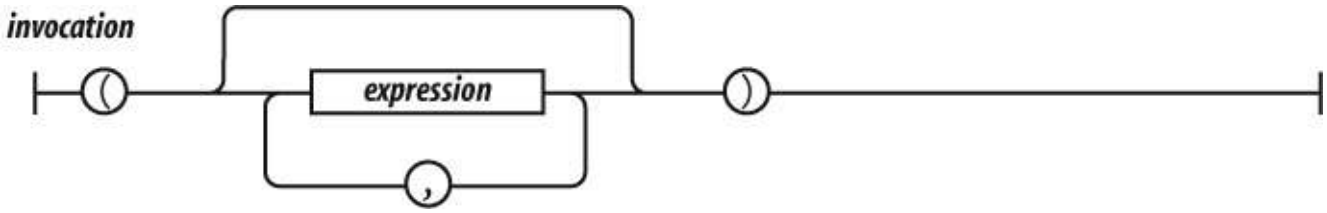


The `+` operator adds or concatenates. If you want it to add, make sure both operands are numbers.

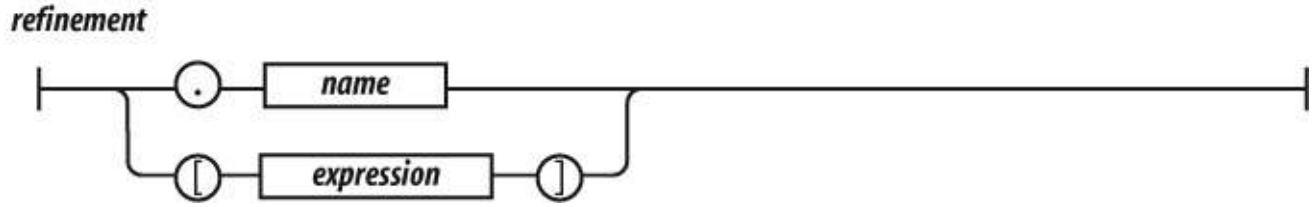
The `/` operator can produce a noninteger result even if both operands are integers.

The `&&` operator produces the value of its first operand if the first operand is falsy. Otherwise, it produces the value of the second operand.

The `||` operator produces the value of its first operand if the first operand is truthy. Otherwise, it produces the value of the second operand.



Invocation causes the execution of a function value. The invocation operator is a pair of parentheses that follow the function value. The parentheses can contain arguments that will be delivered to the function. There will be much more about functions in Chapter 4.



A refinement is used to specify a property or element of an object or array. This will be described in detail in the next chapter.

Literals