

# Summary of Contents

---

Preface .....	xxv
1. Hello JavaScript .....	1
2. Programming Basics .....	19
3. Arrays, Logic, and Loops .....	55
4. Functions .....	81
5. Objects .....	111
6. The Document Object Model .....	143
7. Events .....	183
8. Forms .....	207
9. The Window Object .....	241
10. Testing and Debugging .....	263
11. Functional JavaScript .....	291
12. Object-oriented Programming in JavaScript .....	327
13. Ajax .....	377
14. HTML5 APIs .....	399
15. Organizing Your Code .....	427
16. Next Steps .....	461





# JAVASCRIPT: NOVICE TO NINJA

BY DARREN JONES

# JavaScript: Novice to Ninja

by Darren Jones

Copyright © 2014 SitePoint Pty. Ltd.

**Product Manager:** Simon Mackie

**English Editor:** Kelly Steele

**Technical Editor:** Craig Buckler

**Cover Designer:** Alex Walker

## Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

## Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

## Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood

VIC Australia 3066

Web: [www.sitepoint.com](http://www.sitepoint.com)

Email: [business@sitepoint.com](mailto:business@sitepoint.com)

ISBN 978-0-9924612-2-5 (print)

ISBN 978-0-9924612-1-8 (ebook)

Printed and bound in the United States of America

## About Darren Jones

Darren has been playing around with programming and building websites for over a decade. He wrote the SitePoint book *Jump Start Sinatra*, and also produced the *Getting Started With Ruby* video tutorials for Learnable, as well as writing a number of articles published on SitePoint.

In recent years, having seen just how powerful the language can be, Darren has started to use JavaScript much more heavily. He believes that JavaScript will be the most important programming language to learn in the future.

## About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.



*To my two favourite super heroes,  
Zac & Sienna — love you loads x*





# Table of Contents

---

<b>Preface</b>	xxv
Who Should Read This Book	xxvi
Conventions Used	xxvi
Code Samples	xxvi
Tips, Notes, and Warnings	xxvii
Supplementary Materials	xxviii
Want to Take Your Learning Further?	xxviii
<b>Chapter 1   Hello JavaScript</b>	1
Programming	2
JavaScript	2
The History of JavaScript	3
The Browser Wars	4
Web 2.0	5
Standards	5
HTML5	5
Node.js	6
The Future of JavaScript	6
A Ninja Programming Environment	7
JavaScript Version	7
Text Editors	7
Browser Console	8
Your First JavaScript Program	9
JavaScript in the Browser	10
Three Layers of the Web	10
Unobtrusive JavaScript	11
Graceful Degradation and Progressive Enhancement	12

Your Second JavaScript Program .....	13
The Project: Quiz Ninja .....	15
Chapter Summary .....	17

## **Chapter 2    Programming Basics** .....

Comments .....	20
JavaScript Grammar .....	21
Data Types .....	22
Strings .....	23
Variables .....	24
Reserved Words .....	26
Assignment .....	27
String Properties and Methods .....	27
Numbers .....	31
Octal and Hexadecimal Numbers .....	31
Exponential Notation .....	32
Number Methods .....	32
Arithmetic Operations .....	33
Changing Variables .....	34
Incrementing Values .....	35
Infinity .....	36
NaN .....	37
Type Coercion .....	37
Converting Between Strings and Numbers .....	38
Undefined .....	41
Null .....	41
Booleans .....	41
Logical Operators .....	42
! (Logical NOT) .....	42
&& (Logical AND) .....	43

(Logical OR) .....	44
Lazy Evaluation .....	44
Bitwise Operators .....	45
Bitwise NOT .....	45
Bitwise AND .....	46
Bitwise OR .....	46
Bitwise XOR .....	47
Bitwise Shift Operators .....	47
Comparison .....	48
Equality .....	48
Soft Equality .....	48
Hard Equality .....	49
Inequality .....	51
Greater Than and Less Than .....	51
Quiz Ninja Project .....	53
Summary .....	54

## **Chapter 3**    **Arrays, Logic, and Loops** .....

Arrays .....	56
Stacks of Pizza .....	56
Adding Values to Arrays .....	57
Creating Array Literals .....	58
Removing Values from Arrays .....	59
Array Properties and Methods .....	60
Pop, Push, Shift, and Unshift .....	61
Merging Arrays .....	64
The <code>join()</code> Method .....	65
Slicing and Splicing .....	65
Reverse .....	67
Sort .....	67

Finding if a Value is in an Array .....	68
Multidimensional Arrays .....	68
Logic .....	69
if Statements .....	69
else Statements .....	70
Ternary Operator .....	71
switch Statements .....	71
Loops .....	72
while Loops .....	73
Infinite Loops .....	74
do ... while Loops .....	74
for Loops .....	75
Nested for Loops .....	76
Looping over Arrays .....	77
Quiz Ninja Project .....	77
Summary .....	79

## **Chapter 4**    **Functions** .....

Defining a Function .....	82
Function Declarations .....	82
Function Expressions .....	82
Function() Constructors .....	83
Invoking a Function .....	84
Return Values .....	87
Parameters and Arguments .....	88
The arguments Variable .....	89
Default Arguments .....	90
Scope .....	92
Global Scope .....	92
Local Scope .....	93

Hoisting .....	95
Variable Hoisting .....	95
Function Hoisting .....	96
Callbacks .....	97
Sorting Arrays .....	99
Improving the mean() Function .....	100
Array Iterators .....	101
forEach() .....	102
map() .....	102
reduce() .....	103
Using map() and reduce() Together .....	104
filter() .....	105
Quiz Ninja Project .....	105
Summary .....	109

## Chapter 5    **Objects** .....

Object Literals .....	112
A Super Example .....	112
Creating Objects .....	113
Accessing Properties .....	114
Calling Methods .....	114
Checking if Properties or Methods Exist .....	115
Finding all the Properties of an Object .....	115
Adding Properties .....	116
Changing Properties .....	117
Removing Properties .....	117
Nested Objects .....	118
Objects as Parameters to Functions .....	118
Built-in Objects .....	120
JSON .....	120

The Math Object .....	122
Mathematical Constants .....	122
Mathematical Operations .....	122
Trigonometric Functions .....	125
Random Numbers .....	127
Experimental Methods .....	128
The Date Object .....	128
Constructor Function .....	128
Getter Methods .....	130
Setter Methods .....	132
The RegExp Object .....	133
Creating Regular Expressions .....	133
RegExp Methods .....	134
Basic Regular Expressions .....	134
Character Groups .....	135
Regular Expression Properties .....	136
Special Characters .....	137
Modifiers .....	137
A Practical Example .....	138
String Methods .....	139
Roll the Dice! .....	139
Quiz Ninja Project .....	140
Summary .....	141

## **Chapter 6    The Document Object Model .....** 143

The Document Object Model .....	144
What is the Document Object Model? .....	144
History of the DOM .....	145
An Example Web Page .....	145
Getting Elements .....	147

Legacy DOM Shortcut Methods .....	149
Getting an Element by Its ID .....	150
Get Elements by Their Tag Name .....	150
Get Elements by Their Class Name .....	151
Query Selectors .....	151
Navigating the DOM Tree .....	153
Finding the Value of a Node .....	155
Getting and Setting Attributes .....	156
Getting an Element's Attributes .....	156
Setting an Element's Attributes .....	157
Classes of an Element .....	158
The <code>className</code> Property .....	158
The <code>classList</code> Property .....	158
Updating the DOM by Creating Dynamic Markup .....	160
Creating an Element .....	160
Creating a Text Node .....	161
Appending Nodes .....	161
Putting It All Together in a Function .....	162
Adding Elements to the Page .....	162
Remove Elements from a Page .....	166
Replacing Elements on a Page .....	167
<code>innerHTML</code> .....	169
Live Collections .....	170
Updating CSS .....	171
Camel Case Properties .....	172
Disappearing Act .....	173
Checking Style Properties .....	174
Quiz Ninja Project .....	176
Adding Some Style .....	179
Summary .....	181

<b>Chapter 7</b>	<b>Events</b>	183
Event Listeners		184
Inline Event Handlers		185
Older Event Handlers		186
Event Listeners		186
Example Code		187
The Event Object		189
Types of Event		190
The Event Target		190
Coordinates of an Event		190
Which Mouse Button Was Pressed?		191
Types of Events		191
Mouse Events		191
Keyboard Events		193
Modifier Keys		195
Touch Events		196
Touch Event Properties		197
Removing Event Listeners		198
Stopping Default Behavior		199
Event Propagation		200
Bubbling		200
Capturing		201
Stopping the Bubbling Phase		202
Event Delegation		203
Quiz Ninja Project		203
Summary		205
 <b>Chapter 8</b>	 <b>Forms</b>	 207
Forms		208
A Searching Example		208



Accessing Form Elements .....	209
Form Properties and Methods .....	210
Form Events .....	211
Submitting a Form .....	212
Retrieving and Changing Values from a Form .....	213
Form Controls .....	216
Input Fields .....	219
Text Input Fields .....	219
Password Input Fields .....	220
Checkbox Input Fields .....	220
Radio Button Input Fields .....	222
Hidden Input Fields .....	223
File Input Fields .....	224
Select Drop-down List .....	226
Text Areas .....	227
Buttons .....	227
I Need a Hero! .....	228
Form Validation .....	229
Quiz Ninja Project .....	234
Summary .....	238

## **Chapter 9    The Window Object** .....

The Browser Object Model .....	242
Going Global .....	242
Dialogs .....	243
Browser Information .....	244
Which Browser? .....	244
Location, Location, Location .....	245
The Browser History .....	247
Controlling Windows .....	248

Screen Information .....	249
The Document Object .....	249
<code>document.write()</code> .....	249
Cookies .....	250
Timing Functions .....	254
Animation .....	255
<code>requestAnimationFrame</code> .....	257
Quiz Ninja Project .....	259
Summary .....	261

## **Chapter 10**   **Testing and Debugging** .....

Errors, Exceptions, and Warnings .....	264
The Importance of Testing and Debugging .....	265
Strict Mode .....	266
Linting Tools .....	267
Feature Detection .....	267
Debugging in the Browser .....	268
The Trusty Alert .....	268
Using the Console .....	269
Debugging Tools .....	269
Error Objects .....	270
Throwing Exceptions .....	271
Exception Handling .....	272
<code>try</code> , <code>catch</code> , and <code>finally</code> .....	272
Tests .....	274
Test-driven Development .....	274
Testing Frameworks .....	275
Jasmine .....	275
Crunching Some Numbers .....	276
Quiz Ninja Project .....	287

Summary .....	289
<b>Chapter 11 Functional JavaScript</b> .....	291
What is Functional Programming? .....	292
Functions that Return Functions .....	296
Function Properties and Methods .....	297
Call and Apply Methods .....	297
Custom Properties .....	299
Callbacks .....	300
Event-driven Asynchronous Programming .....	300
Generalized Functions .....	302
Closures .....	302
Function Scope .....	303
The Ninja Training Temple .....	303
A Basic Closure Example .....	305
Returning a Function .....	305
A Counter Example .....	306
A Functional Example .....	306
Immediately Invoked Function Expressions .....	308
Temporary Variables .....	308
Mimicking Block Scope .....	309
Initialization Code .....	310
Safe Use of Strict Mode .....	311
Creating Self-contained Modules .....	311
Functions that Define and Rewrite Themselves .....	312
Init-Time Branching .....	314
Recursive Functions .....	315
Currying .....	317
A General Curry Function .....	319
Quiz Ninja Project .....	320

Summary .....	325
---------------	-----

## Chapter 12 Object-oriented Programming in JavaScript .....

Object-oriented Programming .....	328
Encapsulation .....	328
Polymorphism .....	328
Inheritance .....	329
Classes .....	462
Constructor Functions .....	329
Prototypal Inheritance .....	334
The Prototype Object .....	335
Finding Out the Prototype .....	336
Own Properties and Prototype Properties .....	337
The Prototype Is Live! .....	338
Overwriting Prototype Properties .....	340
What Should the Prototype Be Used For? .....	340
Public and Private Methods .....	342
Inheritance .....	344
The Prototype Chain .....	344
The Object Constructor Function .....	345
Enumerable Properties .....	345
Polymorphism .....	346
Property Attributes and Descriptors .....	350
Getting and Setting Property Descriptors .....	351
Getters and Setters .....	351
Creating Objects from Objects .....	353
Object-based Inheritance .....	355
Object Prototype Chain .....	357
Adding Methods to Built-in Objects .....	358

Mixins .....	360
Using Mixins to Add Properties .....	364
Using Mixins to Create a <code>copy()</code> Method .....	364
Using the Mixin Method to Add Modular Functionality .....	365
Chaining Functions .....	367
This and That .....	367
Use <code>that = this</code> .....	368
Use <code>bind(this)</code> .....	369
Borrowing Methods from Prototypes .....	369
Borrowing Array Methods .....	370
Quiz Ninja Project .....	371
Summary .....	374

<b>Chapter 13</b> <b>Ajax</b> .....	377
Clients and Servers .....	378
A Brief History of Ajax .....	379
The XMLHttpRequest Object .....	381
<code>readyState</code> .....	381
Opening the Request .....	382
Sending the Request .....	383
Receiving the Response .....	383
Receiving Information .....	385
Sending Information .....	388
FormData .....	390
Ajax Timeouts .....	392
JSON With Padding .....	392
JSONP in Action .....	393
Quiz Ninja Project .....	395
Summary .....	459

<b>Chapter 14</b>	<b>HTML5 APIs</b>	399
The Development of HTML5		400
WHATWG		400
Working Together		400
Adoption		401
Modules		433
The data- Attribute		401
HTML5 APIs		403
HTML5 Web Storage		403
Geolocation		406
Web Workers		408
Multimedia		415
Other APIs		418
Drawing with Canvas		418
Shims and Polyfills		422
Quiz Ninja Project		422
Summary		424
 <b>Chapter 15</b>	 <b>Organizing Your Code</b>	 427
Frameworks		428
DOM Manipulation Example		428
jQuery		429
Advantages and Disadvantages of Frameworks		430
When to Use a Framework		431
Some Useful Frameworks		431
Modules		433
CommonJS Modules		435
Asynchronous Module Definitions		436
Package Managers		437
Browserify		437

Bower .....	438
Ender .....	439
MVC Libraries .....	439
A Quick List Example .....	441
MVC Frameworks .....	445
Templates .....	446
Minification .....	449
Task Runners .....	450
Grunt .....	450
Gulp .....	452
Deploying JavaScript .....	453
Quiz Ninja Project .....	454
Deployment .....	456
Summary .....	459

## **Chapter 16**   **Next Steps** ..... 461

What's Next: ECMA6, Harmony .....	461
Block Scope .....	462
Classes .....	462
Arrow Notation .....	464
Default Parameters .....	464
Promises .....	465
Generators .....	466
Modules Using Export and Import .....	468
Ready to Use Today .....	469
Ninja Skills .....	470
Version Control .....	470
Keep Your Knowledge Up to Date .....	471
Use Common JavaScript Coding Patterns .....	472
Build Things .....	472

JavaScript Development Ideas .....	473
HTML5 Game Development .....	473
Single-page Web Applications .....	474
App Development .....	474
Node.js Development .....	475
And There's More! .....	475
Summary .....	476



# Preface

---

The aim of this book is to introduce you to programming using the JavaScript language, eventually helping you to develop into a JavaScript ninja.

This is an exciting time to be learning JavaScript, having finally outgrown its early reputation as a basic scripting language used to produce cringeworthy effects on web pages. Today, JavaScript is used to produce professional and powerful web applications. Modern browsers are now capable of running JavaScript code at lightning speed, and Node.js has helped to revolutionize it by facilitating its use in other environments. This has led to a much more professional and structured approach to building JavaScript applications, where it is now considered a full-fledged programming language. In short, JavaScript has grown up.

JavaScript has a number of cool features that make it stand out from other languages, such as callbacks, first-class functions, prototypal inheritance, and closures. Its event-based model also makes it a very good choice for modern web application development. JavaScript's ace in the pack, though, is something of which every language is envious—its *ubiquity*. JavaScript is available almost everywhere; anybody who has access to a browser can use it. And this is increasing every year as it becomes more readily available outside the browser environment. This translates into JavaScript's reach being immense: it is already the most popular language on GitHub.<sup>1</sup> I can only see JavaScript growing even more popular in the future as it becomes the language of choice for the Internet of Things<sup>2</sup>—helping to control household appliances, even program robots.

Before I get carried away, though, I should point out that JavaScript is far from perfect, having a number of flaws. It is missing some important programming constructs, such as modules and private functions, that are considered standard in many modern programming languages. Yet it's also an unbelievably flexible language, where many of these gaps can be filled using the tools that it provides. In addition, many libraries have sprung into existence that help to extend JavaScript so that it's now able to reach its full potential.

---

<sup>1</sup> <https://www.asad.pw/blog/2014/11/04/github-language-popularity-statistics/>

<sup>2</sup> [http://en.wikipedia.org/wiki/Internet\\_of\\_Things](http://en.wikipedia.org/wiki/Internet_of_Things)

This book starts off with the basics, assuming no programming or JavaScript knowledge, but quickly gets up to speed covering all the main topics in great depth such as functions, objects, and DOM manipulation. More advanced topics such as error handling and testing, functional programming, and OOP are then introduced after the basics have been covered. There have been some exciting new developments in the world of JavaScript over the last few years such as Ajax, HTML5 APIs, and task runners, and these are covered in the last part of the book. There's also a practical project to build a quiz application that is developed throughout the book towards the end of each chapter. I've written with developing for modern browsers in mind, so I've always tried to use the most up-to-date methods in the examples. Having said that, I've also tried to acknowledge if something might not work in an older browser, or if a workaround is needed.

It's a long way ahead—16 chapters, to be precise. But remember, every ninja's journey starts with a single page (or something like that, anyway). So, turn the page and let's get started!

## Who Should Read This Book

---

This book is suitable for beginner-level web designers and developers. Some knowledge of HTML and CSS is assumed, but no previous programming experience is necessary.

## Conventions Used

---

You'll notice that we've used certain typographic and layout styles throughout this book to signify distinct types of information. Look out for the following items.

## Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code is to be found in the book's code archive, the name of the file will appear at the top of the program listing:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css (*excerpt*)

```
border-top: 1px solid #333;
```

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

```
function animate() {
  new_variable = "Hello";
}
```

Where existing code is required for context, rather than repeat all of it, a `:` will be displayed:

```
function animate() {
  :
  return new_variable;
}
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➞ indicates a line break that exists for formatting purposes only, and should be ignored.

```
URL.open("http://www.sitepoint.com/responsive-web-design-real-user-
➞testing/?responsive1");
```

## Tips, Notes, and Warnings



### Hey, You!

Tips will give you helpful little pointers.



### Ahem, Excuse Me ...

Notes are useful asides that are related, but not critical, to the topic at hand. Think of them as extra tidbits of information.



### Make Sure You Always ...

... pay attention to these important points.



### Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

## Supplementary Materials

---

<http://www.learnable.com/books/jsninja1/>

The book's website, which contains links, updates, resources, and more.

<https://github.com/spbooks/jsninja1/>

The downloadable code archive for this book.

<http://community.sitepoint.com/category/javascript>

SitePoint's forums, for help on any tricky web problems.

[books@sitepoint.com](mailto:books@sitepoint.com)

Our email address, should you need to contact us for support, to report a problem, or for any other reason.

## Want to Take Your Learning Further?

---

Thanks for buying this book—we appreciate your support. Do you want to continue learning? You can now gain unlimited access to courses and ALL SitePoint books at Learnable for one low price. Enroll now and start learning today! Join Learnable and you'll stay ahead of the newest technology trends: <http://www.learnable.com>.

# Chapter 2

## Programming Basics

---

In the last chapter, we introduced JavaScript and set up a programming environment where we got our hands dirty with a few JavaScript programs. In this chapter, we're going to delve further to learn how JavaScript works, and start writing some programs.

We'll cover the following topics:

- the importance of well-commented code
- JavaScript grammar—expressions, statements, semicolons, and whitespace
- primitive data types
- strings—string literals and string methods such as `length`
- variables—declaring and assigning
- numbers—decimal, hexadecimal, octal and exponent form, Infinity, and NaN
- arithmetic operations such as `+`, `-`, `*`, `/`, and `%`

- undefined and null
- Booleans—truthy and falsy values
- logical operators—AND, OR, and NOT
- our project—where we'll set some question and answer variables and use alert boxes to display them

## Comments

---

Our first task on our journey to becoming a JavaScript ninja is learning how to write comments in JavaScript. This may seem a strange place to start, because in programming a **comment** is a piece of code that is ignored by the language—it doesn't do anything. Despite this, comments are extremely important: well-commented code is the hallmark of a ninja programmer. It makes it easier for anybody reading your code to understand what's going on, and that includes you! Believe me, you'll be thankful you commented your code when you come back to read it after a few weeks. You only need to write enough so that it's clear what the code is supposed to do.

In JavaScript there are two types of comment.

Single line comments start with `//` and finish at the end of the line:

```
// this is a short comment
```

Multiline comments start with `/*` and finish with `*/`:

```
/* This is a longer comment  
anything here will be ignored  
This is a useful place to put notes  
*/
```

It's good practice to write comments in your code. There are even utilities that can take your comments and produce documentation from them such as JSDoc Toolkit<sup>1</sup>, Docto<sup>2</sup>, and YUIDoc<sup>3</sup>. You'll see lots of comments throughout the code in this book.

---

<sup>1</sup> <http://code.google.com/p/jsdoc-toolkit/>

<sup>2</sup> <http://jashkenas.github.io/docto/>

<sup>3</sup> <http://yui.github.io/yuidoc/>

# JavaScript Grammar

---

The syntax used by JavaScript is known as a C-style syntax, which is similar to the one used by Java.

A JavaScript program is made up of a series of **statements**. Each statement ends with a new line or semicolon.

Here is an example of two statements, one on each line:

```
a = "Hello World!"  
alert(a)
```

This example could also be written as follows, using semicolons at the end of each statement:

```
a = "Hello World!";alert(a);
```

There's no need to actually use a semicolon to terminate a statement because JavaScript interpreters use a process called **Automatic Semicolon Insertion (ASI)**. This will attempt to place semicolons at the end of lines for you; however, it can be error-prone and cause a number of automated services such as code minifiers and validators to not work properly.

For this reason, it's considered best practice to combine the two and write each statement on a new line, terminated by a semi-colon, like so:

```
a = "Hello World!";  
alert(a);
```

A **block** is a series of statements that are collected together inside curly braces:

```
{  
  // this is a block containing 2 statements  
  var a = "Hello!";  
  alert(a);  
}
```

Blocks do not need to be terminated by a semicolon.

Whitespace (such as spaces, tabs, and new lines) is used to separate the different values in each statement. You can use as much whitespace as required to format your code so that it is neat and easy to read. Examples of this include using spaces to indent nested code and multiple lines to separate blocks of code.

## Data Types

JavaScript has six different types of value. There are five primitive **data types**:

- string
- number
- Boolean
- undefined
- null

Any value that isn't one of the primitive data types listed is an **object** (these are covered in Chapter 5). We'll discuss each primitive value over the next few pages.

JavaScript has a special **operator** called **typeof** for finding out the type of a value.

Here are some examples of the different value types:

```
typeof "hello"  
<< "string"  
  
typeof 10  
<< "number"  
  
typeof true  
<< "boolean"
```



```
typeof { ninja: "turtle" }  
<< "object"
```



## Operators

An operator applies an operation to a value, which is known as the *operand*. A **unary operator** only requires one operand; for example:

```
typeof "hello"
```

The operator is **typeof** and the string **"hello"** is the operand.

A **binary operator** requires two operands; for instance:

```
3 + 5
```

The operator is **+** and the numbers **3** and **5** are the operands. There is also a **ternary operator** that requires three operands, which is covered in the next chapter.

## Strings

A **string** is a collection of letters (or characters, to be more precise). We can create a string literal by writing a group of characters inside quote marks like this:

```
"hello"
```



## String Constructor Function

You can also create a string object using the following constructor function:

```
new String("hello")
```

This will create a new string that is the same as the string literal **"hello"**, although it will be classed as an object rather than a primitive value. For this reason it is preferable to use the string literal notation ... not to mention it requires less typing to use literals!

We can also use single quote marks if we prefer:

```
'hello'
```

If you want to use double quote marks inside a string literal, you need to use single quote marks to enclose the string. And if you want to use an apostrophe in your string, you need to employ double quote marks to enclose the string:

```
"It's me"
```

Another option is to do what's called **escaping** the quotation mark. You place a backslash before the apostrophe so that it appears as an apostrophe inside the string instead of terminating the string:

```
'It\'s me'
```



## Escaping Characters

The backslash is used to escape special characters in strings such as:

- single quote marks \'
- double quote marks \"
- end of line \n
- carriage return \r
- tab \t

If you want to actually write a backslash, you need to escape it with another backslash:

```
"This is a backslash \\  
<< "This is a backslash \"
```

## Variables

Variables are common in programming languages. They are a way of storing a value in memory for later use. In JavaScript, we start by declaring a variable. This is done using the keyword `var`:

```
var a; // declare a variable called a
<< undefined

var message;
<< undefined
```

Notice that the console outputs `undefined`. This is a special JavaScript primitive value that is covered later in the chapter, but it's basically saying that the variable has been created but is yet to be assigned a value.

You don't actually have to declare variables before using them, but as we'll see later, bad things can happen if you choose not to. So remember, a ninja will always declare variables.

You can even declare multiple variables at once:

```
var a,b,c; // 3 variables declared at once
<< undefined
```



## Rules for Naming Variables

When naming variables, you should try to give them sensible names that describe what the variable represents; hence, `answer` is a better variable name than `x`.

A variable name can start with any upper or lower case letter, an underscore (`_`), or dollar symbol (`$`). It can also contain numbers but cannot start with them. Here are some examples:

```
$name
_answer
firstName
last_name
address_line1
```

Variable names are case sensitive, so `ANSWER` is different to `Answer` and `answer`.

When using multiple words for variable names there are two conventions that can be used. Camel case starts with a lowercase letter and then each new word is capitalized:

```
firstNameAndLastName
```

Underscore separates each new word with an underscore:

```
first_name_and_last_name
```

JavaScript's built-in functions use the camel-case notation, but you can choose to use one or the other or a combination of the two when naming variables. What's important for a ninja is to *be consistent*.

## Reserved Words

The following words are *reserved* for use by the language and cannot be used to name variables (or the function parameters and object properties that appear in later chapters):

```
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
```

These words are reserved because many of them are used by the language itself, and you will come across them later in this book.

Many are not used by the language, however; one can only assume they were planned to be used at some point, but never were. There are also a few words *not* reserved that should have been as they are an important part of the language:

```
undefined, NaN, Infinity
```

These are covered later in this chapter. You should also avoid using these words for variable names.

## Assignment

To assign a value to a variable, we use the `=` operator. This example shows how we would set the variable `name` to point to the string literal `"Walter"`:

```
var name; // declare the variable first
<< undefined

name = "Walter"; // assign the variable to a string
<< "Walter"
```

Once the variable has been assigned a value, it is displayed in the console output.

To see the value of a variable, simply enter it in the console. The variable `name` now refers to the string `"Walter"`, so it will behave exactly the same as that string:

```
name;
<< "Walter"

typeof name;
<< "string"
```

This is a useful way of dealing with long strings as it saves us from typing them over and over again. It's also useful if the value stored in the variable is likely to change (hence the name, variable).

You can declare and initialize a variable at the same time:

```
var name = "Jesse";
<< "Jesse"
```

You can also declare and assign values to multiple variables in a single statement:

```
var x = 2, y, z = "Hi!"; // y has only been declared, it's undefined
<< undefined
```

## String Properties and Methods

Primitive values and objects have properties and methods. Properties are information about the object or value, while methods perform an action on the object or value—either to change it or to tell us something about it.



## Object Wrappers

Technically, only objects have properties and methods. JavaScript overcomes this by creating *wrapper objects* for primitive values. This all happens in the background, so for all intents and purposes it appears that primitive values also have properties and methods.

We can access the properties of a string using dot notation. This involves writing a dot followed by the property we are interested in. For example, every string has a `length` property that tells us how many characters are in the string:

```
name = "Heisenberg"; // declare and assign a variable
<< "Heisenberg"

name.length; // call the length method on name
<< 10
```

As you can see, this tells us that there are ten characters in the string stored in the `name` variable.



## Bracket Notation

Another notation you can use to access a primitive value's properties are square brackets:

```
name['length']; // note the property name is in quote marks
<< 10
```

All properties of primitive values are **immutable**, which means that they're unable to be changed. You can try, but your efforts will be futile:

```
name.length;
<< 10

name.length = 7; // try to change the length
<< 7
```

```
name.length; // check to see if it's changed  
<< 10
```

A method is an action that a primitive value or object can perform. To call a method, we use the dot operator [ . ] followed by the name of the method, followed by parentheses (this is a useful way to distinguish between a property and a method—methods end with parentheses). For example, we can write a string in all capital letters using the `toUpperCase()` method:

```
name.toUpperCase();  
<< "HEISENBERG"
```

Or the `toLowerCase()` method, which will write my name in all lower-case letters:

```
name.toLowerCase();  
<< "heisenberg"
```

If you want to know which character is at a certain position, you can use the `charAt()` method:

```
name.charAt(1);  
<< "e"
```

This tells us that the character "e" is at position 1. If you were thinking that it should be "H", this is because the first letter is classed as being at position 0 (you'll find that counting usually starts at zero in programming!).

If you want to find where a certain character or substring appears in a string, we can use the `indexOf()` method:

```
name.indexOf("H");  
<< 0
```

If a character doesn't appear in the string, -1 will be returned:

```
name.indexOf("a");
<< -1
```

If we want the last occurrence of a character or substring, we can use the `lastIndexOf()` method:

```
name.lastIndexOf("e");
<< 7
```

The `concat()` method can be used to concatenate two or more strings together:

```
"JavaScript".concat("Ninja");
<< "JavaScriptNinja"

"Hello".concat(" ", "World", "!");
<< "Hello World!"
```

A shortcut for string concatenation is to use the `+` symbol to add the two strings together:

```
"Java" + "Script" + " " + "Ninja";
<< "JavaScript Ninja"
```

The `trim()` method will remove any whitespace from the beginning and end of a string:

```
"  Hello World  ".trim();
➡ // the space in the middle will be preserved
<< "Hello World"

"\t\t JavaScript Ninja! \r".trim();
➡ // escaped tabs and carriage returns are also removed
<< "JavaScript Ninja!"
```



### Support for `trim()`

The `trim()` method was a relatively recent addition to the collection of string methods so is not supported in older browsers.



# Numbers

Number can be *integers* (whole numbers, such as 3) or *floating point decimals* (often referred to as just "decimals" or "floats", such as 3.14159). For example:

```
typeof 3;  
<< "number"  
  
typeof 3.14159;  
<< "number"
```

As you can see in the examples above, JavaScript doesn't distinguish between integers and floating point decimals—they are both given the type of "number", which is a different approach to most other programming languages. This is set out in the ECMAScript specification, although most JavaScript engines will treat integers and floats differently in the background in order to improve efficiency.



## Number Constructor Function

Just like strings, numbers also have a constructor function:

```
new Number(3)
```

This is much more verbose than simply writing the number 3, which is known as a **number literal**, so it is recommended that you stick to using number literals.

## Octal and Hexadecimal Numbers

If a number starts with a 0x, it is considered to be in hexadecimal (base 16) notation:

```
0xAF; // A represents 10, F represents 15  
<< 175
```

Hexadecimal or "hex" numbers are often used for color codes on the Web. You can read more about them on Wikipedia<sup>4</sup>.

If a number starts with a zero, it is *usually* considered to be in octal (base 8) notation:

---

<sup>4</sup> <http://en.wikipedia.org/wiki/Hexadecimal>

```
047; // 4 eights and 7 units
<< 39
```

Octal numbers are not actually part of the ECMAScript standard, but many JavaScript engines implement this convention.

## Exponential Notation

Numbers can also be represented in exponential notation, which is shorthand for "multiply by 10 to the power of" (you may have heard this referred to as "scientific notation" or "standard form"). Here are some examples:

```
1e6; // means 1 multiplied by 10 to the power 6 (a million)
<< 1000000

2E3; // can also be written as 2E3, 2E+3 and 2e+3
<< 2000
```

Fractional values can be created by using a negative index value:

```
2.5e-3; // means 2.5 multiplied by 10 to the power -3 (0.001)
<< 0.0025
```

## Number Methods

Numbers also have some built-in methods, although you need to be careful when using the dot notation with number literals that are integers because the dot can be confused for a decimal point. There are a few ways to deal with this, which we'll demonstrate with the `toExponential()` method; this returns the number as a string in exponential notation.

Use two dots:

```
5..toExponential(); >> "5e+0"
```

Put a space before the dot:

```
5 .toExponential(); >> "5e+0"
```

Always write integers as a decimal:

```
5.0.toExponential(); >> "5e+0"
```

Place the integer in parentheses:

```
(5).toExponential(); >> "5e+0"
```

Assign the number to a variable:

```
var number = 5;
>> 5

number.toExponential();
>> "5e+0"
```

The `toFixed()` method rounds a number to a fixed number of decimal places:

```
var pi = 3.1415926;
<< undefined

pi.toFixed(3); // only one dot needed when using variables
<< "3.142"
```

Note that the value is returned as a string.

The `toPrecision()` method rounds a number to a fixed number of significant figures that is once again returned as a string (and often using exponential notation):

```
325678..toPrecision(2);
<< "3.3e+5"

2.459.toPrecision(2);
<< "2.5"
```

## Arithmetic Operations

All the usual arithmetic operations can be carried out in JavaScript.

Addition:

```
5 + 4.3;  
<< 9.3
```

Subtraction:

```
6 - 11;  
>> -5
```

Multiplication:

```
6 * 7;  
<< 42
```

Division:

```
3/7;  
<<0.42857142857142855
```

You can also calculate the remainder of a division using the % operator:

```
23%6; // the same as asking 'what is the remainder  
      ➡when 13 is divided by 6'  
<< 5
```

This is similar to, but not quite the same as, modulo arithmetic. That's because the result always has the same sign as the first number:

```
-4%3; // -4 modulo 3 would be 2  
<< -1
```

## Changing Variables

If a variable has been assigned a numerical value, it can be increased using the following operation:

```
points = 0; // initialize points score to zero
<< 0

points = points + 10;
<< 10
```

This will increase the value held in the `points` variable by 10. You can also use the compound assignment operator, `+=`, which is a shortcut for performing the same task, but helps you avoid writing the variable name twice:

```
points += 10;
<< 20
```

There are equivalent compound assignment operators for all the operators in the previous section:

```
points -= 5; // decreases points by 5
<< 15

points *= 2; // doubles points
<< 30

points /= 3; // divides value of points by 3
<< 10

points %= 7; // changes the value of points to the remainder
           ➡ if its current value is divided by 7
<< 3
```

## Incrementing Values

If you only want to increment a value by 1, you can use the `++` operator. This goes either directly before or after the variable.

So what's the difference between putting the `++` operator before or after the variable? The main difference is the value that is returned by the operation. Both operations increase the value of the `points` variable by 1, but `points++` will return the original value *then* increase it by 1, whereas `++points` will increase the value by 1, then return the new value:

```
points++; // will return 3, then increase points to 4
<< 3

++points; // will increase points to 5, then return it
<< 5
```

There is also a -- operator that works in the same way:

```
points--;
<< 5

--points;
<< 3
```

## Infinity

Infinity is a special error value in JavaScript that is used to represent any number that is too big for JavaScript to deal with. The biggest number that JavaScript can handle is 1.7976931348623157e+308:

```
1e308; // 1 with 308 zeroes!
<< 1e308

2e308; // too big!
<< Infinity
```

There is also a value -Infinity, which is used for negative numbers that go below -1.7976931348623157e+308:

```
-1e309;
<< -Infinity
```

The value of Infinity can also be obtained by dividing by zero:

```
1/0;
<< Infinity
```

The smallest number that JavaScript can deal with is 5e-324. Anything below this evaluates to either 5e-324 or zero:

```
5e-324;  
<< 5e-324  
  
3e-325;  
<< 5e-324  
  
2e-325;  
<< 0
```

## NaN

NaN is an error value that is short for "Not a Number". It is used when an operation is attempted and the result isn't numerical:

```
"hello" * 5;  
<< NaN
```

The result returned by the `typeof` operator is rather ironic, however:

```
typeof NaN;  
<< 'number'
```

## Type Coercion

**Type coercion** is the process of converting the type of a value in the background to try and make an operation work. For example, if you try to multiply a string and a number together, JavaScript will attempt to coerce the string into a number:

```
"2" * 8;  
<< 16
```

This may seem useful, but the process is not always logical or consistent, causing a lot of confusion. For example, if you try to *add* a string and a number together, JavaScript will convert the number to a string and then concatenate the two strings together:

```
"2" + 8;  
<< "28"
```

This can make it difficult to spot type errors in your code, so you should always try to be very explicit about the types of values you are working with.

## Converting Between Strings and Numbers

We can convert numbers to strings and vice versa using a variety of methods.

### Converting Strings to Numbers

To convert a string into a number we can multiply a numerical string by 1, which will convert it into a number because of type coercion:

```
answer = "5" * 1;  
<< 5  
  
typeof answer;  
<< "number"
```

Another neat way of converting a string to an integer is to simply place a + symbol in front of it:

```
answer = +"5";  
<< 5  
  
typeof answer;  
<< "number"
```

Yet another way to convert a string into a number is to use the `Number` function:

```
Number("23");  
<< 23
```

This is the preferred way to convert strings to numbers as it avoids type coercion in the background. The conversion is explicit, making it obvious what is being done.



## Converting Numbers to Strings

To change numbers into strings you can add an empty string, which will use type coercion to silently convert the number into a string in the background:

```
3 + '';  
<< "3"
```

The preferred way, however, is to use the `String` function:

```
String(3);  
<< "3"
```

There is also the very similar `toString()` method, but this may change the base of the number. For example, if you want to write the number 10 in binary (base two), you could write:

```
> 10..toString(2):  
<< "1010"
```

You can go up to base 36, although after base ten, letters are used to represent the digits:

```
> 1000000..toString(36) // a million in base 36  
<< "1fls"
```

## Parsing Numbers

There is also a useful function called `parseInt()` that can be used to convert a string representation of a numerical value back into an integer. You can specify the base of the number you are trying to convert, for example:

```
parseInt("1010",2); // converts from binary, back to decimal  
<< 10  
  
parseInt("omg",36);  
<< 31912
```

```
parseInt("23",10);  
<< 23
```

If a string starts with a number, the `parseInt` function will use this number and ignore any letters that come afterwards:

```
var address = "221B Baker Street"  
<< undefined  
  
parseInt(address, 10)  
<< 221
```

If you try to do this with the `Number` function, it returns `NaN`:

```
Number(address)  
<< NaN
```

And if you use `parseInt` with a decimal, it will remove anything after the decimal point:

```
parseInt("2.4",10)  
<< 2
```

Be careful not to think that this is rounding the number to the nearest integer; it simply removes the part after the decimal point, as seen in this example:

```
parseInt("2.9",10)  
<< 2
```

There is also a similar function called `parseFloat()` that converts strings into floating point decimal numbers:

```
parseFloat("2.9",10)  
<< 2.9
```

## Undefined

---

`undefined` is the value given to variables that have not been assigned a value. It can also occur if an object's property doesn't exist or a function has a missing parameter. It is basically JavaScript's way of saying "I can't find a value for this."

## Null

---

`null` means "no value". It can be thought of as a placeholder that JavaScript uses to say "there should be an value here, but there isn't at the moment."

`undefined` and `null` are both "non-value" values. They are similar, although they behave slightly differently. For example, if you try to do sums with them:

```
10 + null // null behaves like zero  
<< 10  
  
10 + undefined // undefined is not a number  
<< NaN
```

`null` is coerced to be 0, making the sum possible whereas `undefined` is coerced to `NaN`, making the sum impossible to perform.

Values tend to be set to `undefined` by JavaScript, whereas values are usually set to `null` manually by the programmer.

## Booleans

---

There are only two Boolean values: `true` and `false`. They are named after George Boole, an English mathematician who worked in the field of algebraic logic. Boolean values are fundamental in the logical statements that make up a computer program. Every value in JavaScript has a Boolean value and most of them are `true` (these are known as 'truthy' values).

To find the Boolean value of something, you can use the `Boolean` function like so:

```
Boolean("hello");  
<< true  
  
Boolean(42);  
<< true  
  
Boolean(0);  
<< false
```

Only seven values are always `false` and these are known as falsy values:

```
* "" // double quoted empty string  
* '' // single quoted empty string  
* 0  
* NaN  
* false  
* null  
* undefined
```



### Truthy and Falsy Values

The fact that empty strings and zero are considered falsy can cause confusion at times, especially since other programming languages don't behave similarly. A ninja needs to be especially careful when dealing with numbers that might be zero, or strings that are empty.

For more on truthy and falsy values, see this article on SitePoint.<sup>5</sup>

## Logical Operators

A logical operator can be used with any primitive value or object. The results are based on whether the values are considered to be truthy or falsy.

### ! (Logical NOT)

Placing the `!` operator in front of a value will convert it to a Boolean and return the opposite value. So truthy values will return `false`, and falsy values will return `true`. This is known as *negation*:

---

<sup>5</sup> <http://www.sitepoint.com/javascript-truthy-falsy/>

```
!true;
<< false

!0;
<< true
```

You can use double negation (!!) to find out if a value is truthy or falsy (it is a shortcut to using the `Boolean` function we employed earlier because you are effectively negating the negation):

```
!!'';
<< false

!!"hello";
<< true

!!3;
<< true

!!NaN;
<< false

!!"false";
<< true

!!'0';
<< true
```

## && (Logical AND)

Imagine that you are having a party and want to have some rules about who is allowed in. You might want to only allow people who are wearing glasses AND who are over 18 to be allowed in. This is an example of a logical AND condition: anybody coming to the party must satisfy *both* conditions before they are let in.

The logical AND operator works on two or more values (the operands) and only evaluates to `true` if *all* the operands are truthy. The value that is returned is the *last* truthy value if they are all true, or the *first* falsy value if at least one of them is false:

```
true && true;  
<< true  
  
3 && 0; // returns 0 because it is falsy  
<< 0
```

## || (Logical OR)

Now imagine that you relax the rules for your party and allow people in if they wear glasses *OR* are over 18. This means that they only have to satisfy one of the rules to be allowed in—an example of a logical OR condition.

The logical OR operator also works on two or more operands, but evaluates to true if *any* of the operands are true, so it only evaluates to false if both operands are falsy. The value that is returned is the *first* truthy value if any of them are true, or the *last* falsy value if all of them are false:

```
true || false;  
<< true  
  
NaN || undefined;  
➡ // both NaN and undefined are falsy, so undefined will be returned  
<< undefined
```

## Lazy Evaluation

Remember the party example when the condition for entry was that attendees had to wear glasses *and* be over 18? If you saw somebody without glasses, would you bother asking them to prove that they were over 18? There'd be no point because by not wearing glasses, they wouldn't be allowed in anyway.

When the rules were relaxed, people were allowed in if they were wearing glasses *or* if over 18. If somebody arrived wearing glasses, there would be no need to check their age.

These are examples of **lazy evaluation**—you only check as many conditions as you have to for somebody to be allowed in. JavaScript performs a similar task and uses lazy evaluation when processing the logical AND and OR operators. This means that it stops evaluating any further operands once the result is clear.

For example, for a logical AND expression to be true, all the operands have to be true; if any of them are false, there is no point checking any subsequent operands as the result will still be false. Similarly, for a logical OR to be true, only one of the operands has to be true; hence, as soon as an operand is evaluated to true, the result is returned as true and any subsequent operands won't be checked as the result is of no consequence.

This is demonstrated in the examples below:

```
a = 0; // declare the variable a and assign the value of 0
<< 0

false && (a = 1); // (a = 1) is truthy, but it won't be evaluated,
➡ since the first operand is false
<< false

a; // the value of a is still 0
<< 0

false || (a = 1); // this will evaluate both operands, so a will be
➡ assigned the value of 1, which is returned
<< 1
```

## Bitwise Operators

Bitwise operators work with operands that are 32-bit integers. These are numbers written in binary (base two) that have 32 digits made up of just 0s and 1s. Here are some examples:

```
5 is written as   00000000000000000000000000000101
100 is written as 00000000000000000000000000001100100
15 is written as  000000000000000000000000000001111
```

JavaScript will convert any values used with bitwise operators into a 32-bit integer and then carry out the operation.

## Bitwise NOT

The bitwise NOT operator [~] will convert the number to a 32-bit integer, then change all the 1s to 0 and all the 0s to 1s. For example, 2476 can be represented as:

000000000000000000001011010101100

Which will change to:

111111111111111110100101010011

This is 1073736019, but the result actually uses negative values, as you can see in the code:

~44;  
<< -45

In most cases, this operator will return an integer that adds to the original operand to make -1.

## Bitwise AND

You can also use the bitwise AND operator, `&`, which will convert both numbers into binary and returns a number that in binary has a 1 in each position for which the corresponding bits of both operands are 1s. Here's an example:

```
12 & 10; // in binary this is 1100 & 1010, so only the first digit
↳ is 1 in both cases
<< 8
```

It can also be used with non-integers, where it returns 1 for true and 0 for false.

```
5 & "hello"; // both are true
<< 1
```

## Bitwise OR

There is also the bitwise OR operator, `[|]`, which will convert both numbers into binary and return a number that in binary has a 1 in each position for which the corresponding bits of either operands are 1s. Here's an example:



```
12 | 10; // in binary this is 1100 & 1010, so the first 3 digits
      ↳ contain a 1
      << 14
```

This can also be used with non-integers, and returns 1 for true and 0 for false.

```
' ' | "";
<< 0 // both are falsy
```

## Bitwise XOR

Another operation is the bitwise XOR operator, `[^]`, which stands for "eXclusive OR". This will convert both numbers into binary and return a number that in binary has a 1 in each position for which the corresponding bits of either operands are 1s, but not both 1s. Here's an example:

```
12 ^ 10; // in binary this is 1100 & 1010, so only the second and
      ↳ third digits are exclusively 1s
      << 6
```

When using non-integer values, this evaluates to 1 if *either* operands are truthy and evaluates to 0 if both operands are truthy or both are falsy:

```
1 ^ 0; // The first operand is truthy
<< 1

true ^ true; // if both operands are true then the result is false
<< 0
```

## Bitwise Shift Operators

The bitwise shift operators, `<<` and `>>`, will move the binary representation a given number of places to the right or left, which effectively multiplies or divides the number by powers of two:

```
3 << 1; // multiply by 2
<< 6

16 >> 1; // divide by 2
<< 8
```

```
5 << 3; multiply by 2 cubed (8)
<< 40
```

## Comparison

We often need to compare values when we are programming. JavaScript has several ways to compare two values.

### Equality

Remember earlier, when we assigned a value to a variable? We used the = operator to do this, which would be the logical choice for testing if two values are equal.

Unfortunately, we can't use it because it is used for assigning values to variables. For example, say we had a variable called `answer` and we wanted to check if it was equal to 5, we might try doing this:

```
answer = 5;
<< 5
```

What we've actually done is *assign* the value of 5 to the variable `answer`, effectively overwriting the previous value!

The correct way to check for equality is to use either a double equals operator, `==`, known as "soft equality" or the triple equals operator, `===`, known as "hard equality".

### Soft Equality

We can check if `answer` is in fact equal to 5 using soft equality, like so:

```
answer == 5;
<< true
```

This seems to work fine, but unfortunately there are some slight problems when using soft equality:

```
answer == "5";  
<< true
```

As you can see, JavaScript is returning `true` when we are checking if the variable `answer` is equal to the *string* `"5"`, when in fact `answer` is equal to the *number* `5`. This is an important difference, but when a soft inequality is used, JavaScript will attempt to coerce the two values to the same type when doing the comparison. This can lead to some very strange results:

```
" " == 0;  
<< true  
  
" " == "0";  
<< false  
  
false == "0";  
<< true  
  
"1" == true;  
<< true  
  
"2" == true;  
<< false  
  
"true" == true;  
<< false  
  
null == undefined;  
<< true
```

As you can see, values that are not actually equal have a tendency to be reported as being equal to each other when using the soft equality operator.

## Hard Equality

Hard equality also tests that the two values are the same type:

```
answer === 5;  
<< true  
  
answer === "5";  
<< false
```

```
null === undefined;  
<< false
```

As you can see, hard equality reports that the variable `answer` is the number 5, but not the string "5". It also correctly reports that `null` and `undefined` are two different values.



### When is Not a Number not Not a Number?

The only strange result produced by hard equality is this:

```
NaN === NaN;  
<< false
```

NaN is the only value in JavaScript that is not equal to itself. To deal with this, there is a special function called `isNaN` to test it:

```
isNaN(NaN);  
<< true  
  
isNaN(5);  
<< false
```

Unfortunately, this doesn't always work properly, as can be seen in this example:

```
isNaN("hello");  
<< true
```

This is because the function first of all tries to convert the string to a number, and strings without numerals are converted to NaN:

```
Number("hello");  
<< NaN
```

The only way to accurately check if a value is NaN is to check that its type is a number (because NaN is of type "number") and also check that the `isNaN` function returns `true`. These two conditions should be combined using the logical AND (`&&`) that we saw earlier:

```

isnan = NaN; // set the variable isnan to be NaN
<< NaN

notnan = "hello"; // set the variable notnan to "hello"
<< "hello"

typeof(isnan) === "number" && isNaN(isnan);
<< true

typeof(notnan) === "number" && isNaN(notnan);
<< false

```

So, a JavaScript ninja should always use hard equality when testing if two values are equal. This will avoid the problems caused by JavaScript's type coercion.

If you want to check whether a number represented by a string is equal to a number, you should convert it to a number yourself explicitly:

```

> Number("5") === 5
<< true

```

This can be useful when you're checking values entered in a form as these are always strings.

## Inequality

We can check if two values are *not* equal using the inequality operator. There is a soft inequality operator, `!=` and a hard inequality operator, `!==`. These work in a similar way to the soft and hard equality operators:

```

16 != "16"; // type coercion makes these equal
<< false

16 !== "16";
<< true

```

As with equality, it is much better to use the hard inequality operator as this will give more reliable results unaffected by type coercion.

## Greater Than and Less Than

We can check if a value is greater than another using the `>` operator:

```
8 > 4; << true
```

You can also use the "less than" < operator in a similar way:

```
8 < 4; << false
```

If you want to check if a value is greater than *or equal* to another value, you can use the >= operator, but be careful, the equality test works in the same way as the soft equality operator:

```
8 >= 4;  
<< true  
  
8 >= 8;  
<< true  
  
8 >= "8";  
<< true
```

As you can see, type coercion means that strings can be confused with numbers. Unfortunately, there are no "hard" greater-than or equal-to operators, so an alternative way to avoid type coercion is to use a combination of the greater-than operator, logical OR, and a hard equality:

```
8 > 8 || 8 === 8;  
<< true  
  
8 > "8" || 8 === "8";  
<< false
```

There is also a similar "less-than or equal-to" operator:

```
-1 <= 1;  
<< true  
  
-1 <= -1;  
<< true
```

These operators can also be used with strings, which will be alphabetically ordered to check if one string is "less than" the other:

```
"apples" < "bananas";
>> true
```

Be careful, though, as the results are case-sensitive and upper-case letters are considered to be "less than" lower-case letters:

```
"apples" < "Bananas";
>> false
```

## Quiz Ninja Project

Now that we have come to the end of the chapter, it's time to put what we've learned into practice in our Quiz Ninja project.

Since we've been learning all about JavaScript in this chapter, we're going to add some code in the **scripts.js** file. Open that file and add the following lines:

*scripts.js (excerpt)*

```
var question = "What is Superman's real name?"
var answer = prompt(question);
alert("You answered " + answer);
```

Now let's go through this code line by line to see what is happening:

```
var question = "What is Superman's real name?"
```

This declares a variable called `question` and assigns the string "What is Superman's real name?" to it. Next, we need to ask the question stored in the `question` variable, using a prompt dialog:

```
var answer = prompt(question);
```

A **prompt dialog** allows the player to type in an answer, which is stored in a variable called `answer`.

Finally, we use an alert dialog to display the player's answer:

```
alert("You answered " + answer);
```

This shows the player the answer they provided. In the next chapter we'll look at how to check if it's correct.

Have a go at playing the quiz by opening the **index.htm** file in your browser. It should look a little like the screenshot in Figure 2.1.

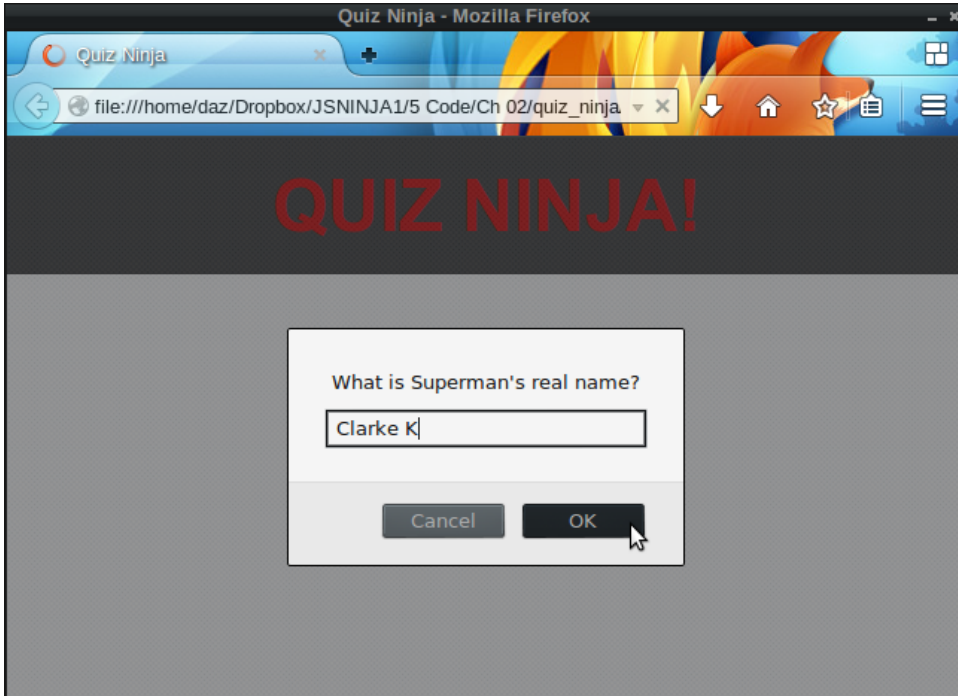


Figure 2.1. Let's play Quiz Ninja!

This is a good example of using the prompt and alert dialogs, along with variables to store the responses in to create some interactivity with the user.

## Summary

In this chapter, we've learned about the primitive data types that are the basic building blocks of all JavaScript programs: strings, numbers, Booleans, undefined and null. We've also learned about variables and different methods for strings and numbers, as well as how to convert between the two types. We finished by looking at the different logical operators and ways of comparing values.

In the next chapter, we'll be looking at arrays, logic, and loops.