Mastering JavaScript

JavaScript is a high-level, dynamic, untyped, and lightweight programming language. The majority of websites employ JavaScript, and it is well supported by all modern web browsers without plugins. However, the JavaScript landscape has changed dramatically in recent years, and you need to adapt to the new world of JavaScript that people now expect. This book will be your companion as you master JavaScript and build innovative web applications. To begin with, you will become familiar with the language constructs and making code easy to organize. You will gain a concrete understanding of variable scoping, loops, and best practices for using types and data structures, as well as the coding style and recommended code organization patterns in JavaScript. You will graduate from intermediate-level skills to advanced techniques as you come to understand crucial language concepts and design principles. By the end of the book, you will understand how reactive JavaScript is going to be the new paradigm.

Who this book is written for

This book is ideal for web developers who are looking to master modern JavaScript concepts and design principles. You should already have an intermediate level of JavaScript knowledge before starting this book.

What you will learn from this book

- Get a run-through of the basic JavaScript language constructs
- Get familiar with the functions and closures of JavaScript
- Explore regular expressions in JavaScript
- Code using the powerful object-oriented features in JavaScript
- Test and debug your code using JavaScript strategies
- Master DOM manipulation, cross-browser strategies, and ES6
- Understand the basic concurrency constructs in Javascript and best performance strategies
- Learn to build scalable server application in JavaScript using Node.js

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'JavaScript Primer'
- A synopsis of the book’s content
- More information on Mastering JavaScript
About the Author

Ved Antani is an AVP (engineering) at Myntra. Before Myntra, he worked with Electronic Arts, NetApp, and Oracle. Ved is passionate about programming and has been programming in JavaScript since 2005. He has extensive experience in building scalable systems and mobile applications. Ved is a minimalist, a father, and an avid tea drinker.
Preface

It would seem that everything that needs to be written about JavaScript has been written. Frankly, it is difficult to find a topic related to JavaScript that has not been discussed ad nauseam. However, JavaScript is changing at a rapid pace. ECMAScript 6 has the potential to transform the language and how we code in it. Node.js has already changed the way in which we write servers in JavaScript. Newer ideas such as React and Flux will drive the next iteration of the language. While we spend our time learning the new features, there is no denying that the foundational ideas of JavaScript have to be mastered. These ideas are fundamental and need attention. If you are already an experienced JavaScript developer, you will realize that modern JavaScript is vastly different from the language that most people have known. Modern JavaScript demands a specific stylistic discipline and rigor of thought. Tools are more powerful and slowly becoming an integral part of the development workflow. Though the language seems to be changing, it is built on some very solid and constant ideas. This book emphasizes on these fundamental ideas.

While the book was being written, things kept changing in the JavaScript landscape. Luckily, we were able to include all the important and relevant updates in this book.

*Mastering JavaScript* provides you with a detailed overview of the language's fundamentals and some of the modern tools and libraries, such as jQuery, Underscore.js, and Jasmine.

We hope that you enjoy this book as much as we enjoyed writing it.
What this book covers

Chapter 1, JavaScript Primer, focuses on the language constructs without spending too much time on the basic details. We will cover the trickier parts of variable scoping and loops and best practices for using types and data structures. We will also cover a lot of ground on the coding style and recommended code organization patterns.

Chapter 2, Functions, Closures and Modules, covers the core of the language intricacies. We will discuss the complexities involved in using functional aspects with different treatment for closures in JavaScript. This is a careful and elaborate discussion that will prepare you to explore more advanced design patterns further on.

Chapter 3, Data Structures and Manipulation, takes a detailed look at regular expressions and arrays. Arrays are a fundamental data type in JavaScript and this chapter will help you work effectively with arrays. Regular expressions can make your code concise—we will take a detailed look at how to use RegEx effectively in your code.

Chapter 4, Object-Oriented JavaScript, discusses object orientation in JavaScript. We will discuss inheritance and the prototype chain and focus on understanding the prototypical inheritance model that JavaScript offers. We will also discuss how different this model is from other object-oriented models to help Java or C++ programmers get familiarized with the change.

Chapter 5, JavaScript Patterns, discusses common design patterns and how to implement them in JavaScript. Once you master the object-oriented model for JavaScript, it is easier to understand design and programming patterns to write modular and easy-to-maintain code.

Chapter 6, Testing and Debugging, covers various modern methods to test and debug issues in JavaScript code. We will also explore continuous testing and test-driven methodologies for JavaScript. We will use Jasmine as the test framework.

Chapter 7, ECMA Script 6, focuses on the newer language features introduced by ECMAScript 6 (ES6). It makes JavaScript more powerful and this chapter will help you understand the newer features and how to use them in your code.

Chapter 8, DOM Manipulation and Events, takes a detailed look at JavaScript as a language of the browser. This chapter discusses DOM manipulation and browser events.

Chapter 9, Server-Side JavaScript, explains how we can use Node.js to write scalable server systems in JavaScript. We will discuss the architecture of Node.js and several useful techniques.
1

JavaScript Primer

It is always difficult to pen the first few words, especially on a subject like JavaScript. This difficulty arises primarily because so many things have been said about this language. JavaScript has been the Language of the Web—lingua franca, if you will, since the earliest days of the Netscape Navigator. JavaScript went from a tool of the amateur to the weapon of the connoisseur in a shockingly short period of time.

JavaScript is the most popular language on the web and open source ecosystem. http://github.info/ charts the number of active repositories and overall popularity of the language on GitHub for the last few years. JavaScript's popularity and importance can be attributed to its association with the browser. Google's V8 and Mozilla's SpiderMonkey are extremely optimized JavaScript engines that power Google Chrome and Mozilla Firefox browsers, respectively.

Although web browsers are the most widely used platforms for JavaScript, modern databases such as MongoDB and CouchDB use JavaScript as their scripting and query language. JavaScript has become an important platform outside browsers as well. Projects such as Node.js and io.js provide powerful platforms to develop scalable server environments using JavaScript. Several interesting projects are pushing the language capabilities to its limits, for example, Emscripten (http://kripken.github.io/emscripten-site/) is a Low-Level Virtual Machine (LLVM)-based project that compiles C and C++ into highly optimizable JavaScript in an asm.js format. This allows you to run C and C++ on the web at near native speed.

JavaScript is built around solid foundations regarding, for example, functions, dynamic objects, loose typing, prototypal inheritance, and a powerful object literal notation.
While JavaScript is built on sound design principles, unfortunately, the language had to evolve along with the browser. Web browsers are notorious in the way they support various features and standards. JavaScript tried to accommodate all the whims of the browsers and ended up making some very bad design decisions. These bad parts (the term made famous by Douglas Crockford) overshadowed the good parts of the language for most people. Programmers wrote bad code, other programmers had nightmares trying to debug that bad code, and the language eventually got a bad reputation. Unfortunately, JavaScript is one of the most misunderstood programming languages (http://javascript.crockford.com/javascript.html).

Another criticism leveled at JavaScript is that it lets you get things done without you being an expert in the language. I have seen programmers write exceptionally bad JavaScript code just because they wanted to get the things done quickly and JavaScript allowed them to do just this. I have spent hours debugging very bad quality JavaScript written by someone who clearly was not a programmer. However, the language is a tool and cannot be blamed for sloppy programming. Like all crafts, programming demands extreme dedication and discipline.

A little bit of history

In 1993, the Mosaic browser of National Center for Supercomputing Applications (NCSA) was one of the first popular web browsers. A year later, Netscape Communications created the proprietary web browser, Netscape Navigator. Several original Mosaic authors worked on Navigator.

In 1995, Netscape Communications hired Brendan Eich with the promise of letting him implement Scheme (a Lisp dialect) in the browser. Before this happened, Netscape got in touch with Sun Microsystems (now Oracle) to include Java in the Navigator browser.

Due to the popularity and easy programming of Java, Netscape decided that a scripting language had to have a syntax similar to that of Java. This ruled out adopting existing languages such as Python, Tool Command Language (TCL), or Scheme. Eich wrote the initial prototype in just 10 days (http://www.computer.org/csdl/mags/co/2012/02/mco2012020007.pdf), in May 1995. JavaScript's first code name was Mocha, coined by Marc Andreessen. Netscape later changed it to LiveScript, for trademark reasons. In early December 1995, Sun licensed the trademark Java to Netscape. The language was renamed to its final name, JavaScript.
How to use this book

This book is not going to help if you are looking to get things done quickly. This book is going to focus on the correct ways to code in JavaScript. We are going to spend a lot of time understanding how to avoid the bad parts of the language and build reliable and readable code in JavaScript. We will skirt away from sloppy features of the language just to make sure that you are not getting used to them—if you have already learned to code using these habits, this book will try to nudge you away from this. There will be a lot of focus on the correct style and tools to make your code better.

Most of the concepts in this book are going to be examples and patterns from real-world problems. I will insist that you code each of the snippets to make sure that your understanding of the concept is getting programmed into your muscle memory. Trust me on this, there is no better way to learn programming than writing a lot of code.

Typically, you will need to create an HTML page to run an embedded JavaScript code as follows:

```html
<!DOCTYPE html>
<html>
<head>
  <script type="text/javascript" src="script.js"></script>
  <script type="text/javascript">
    var x = "Hello World";
    console.log(x);
  </script>
</head>
<body>
</body>
</html>
```

This sample code shows two ways in which JavaScript is embedded into the HTML page. First, the `<script>` tag in `<head>` imports JavaScript, while the second `<script>` tag is used to embed inline JavaScript.

Downloading the example code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
You can save this HTML page locally and open it in a browser. On Firefox, you can open the **Developer** console (Firefox menu | **Developer** | **Web Console**) and you can see the "Hello World" text on the **Console** tab. Based on your OS and browser version, the screen may look different:

![Console screenshot](image)

You can run the page and inspect it using Chrome's **Developer Tool**:

![Developer Tool screenshot](image)

A very interesting thing to notice here is that there is an error displayed on the console regarding the missing `.js` file that we are trying to import using the following line of code:

```html
<script type="text/javascript" src="script.js"></script>
```

Using browser developer consoles or an extension such as **Firebug** can be very useful in debugging error conditions in the code. We will discuss in detail the debugging techniques in later chapters.

Creating such HTML scaffolds can be tedious for every exercise in this book. Instead, we want to use a **Read-Eval-Print-Loop (REPL)** for JavaScript. Unlike Python, JavaScript does not come packaged with an REPL. We can use Node.js as an REPL. If you have Node.js installed on your machine, you can just type `node` on the command line and start experimenting with it. You will observe that Node REPL errors are not very elegantly displayed.
Let's see the following example:

```
EN-VedA:~$ node
> function greeter(){
  x="World"
SyntaxError: Unexpected identifier
  at Object.exports.createScript (vm.js:44:10)
  at REPLServer.defaultEval (repl.js:117:23)
  at bound (domain.js:254:14)
```

After this error, you will have to restart. Still, it can help you try out small fragments of code a lot faster.

Another tool that I personally use a lot is JS Bin (http://jsbin.com/). JS Bin provides you with a great set of tools to test JavaScript, such as syntax highlighting and runtime error detection. The following is a screenshot of JS Bin:

```
function test(){
  var undeterminedValue = "elephant";
  if (isNaN(parseInt(undeterminedValue,2))) {
    console.log("handle not a number case");
  } else {
    console.log("handle number case");
  }
}
test();
```

Based on your preference, you can pick the tool that makes it easier to try out the code samples. Regardless of which tool you use, make sure that you type out every exercise in this book.
Hello World
No programming language should be published without a customary Hello World program—why should this book be any different?

Type (don't copy and paste) the following code in JS Bin:

```javascript
function sayHello(what) {
    return "Hello " + what;
}

console.log(sayHello("world"));
```

Your screen should look something as follows:

![JavaScript Bin Example](image)

An overview of JavaScript
In a nutshell, JavaScript is a prototype-based scripting language with dynamic typing and first-class function support. JavaScript borrows most of its syntax from Java, but is also influenced by Awk, Perl, and Python. JavaScript is case-sensitive and white space-agnostic.

Comments
JavaScript allows single line or multiple line comments. The syntax is similar to C or Java:

```javascript
// a one line comment

/* this is a longer,
   multi-line comment */

/* You can't /* nest comments */ SyntaxError */
```
Variables

Variables are symbolic names for values. The names of variables, or identifiers, must follow certain rules.

A JavaScript variable name must start with a letter, underscore (_), or dollar sign ($); subsequent characters can also be digits (0-9). As JavaScript is case sensitive, letters include the characters A through Z (uppercase) and the characters a through z (lowercase).

You can use ISO 8859-1 or Unicode letters in variable names.

New variables in JavaScript should be defined with the var keyword. If you declare a variable without assigning a value to it, its type is undefined by default. One terrible thing is that if you don't declare your variable with the var keyword, they become implicit globals. Let me reiterate that implicit globals are a terrible thing—we will discuss this in detail later in the book when we discuss variable scopes and closures, but it's important to remember that you should always declare a variable with the var keyword unless you know what you are doing:

```javascript
var a;      // declares a variable but its undefined
var b = 0;
console.log(b);   // 0
console.log(a);   // undefined
console.log(a+b); // NaN
```

The NaN value is a special value that indicates that the entity is not a number.

Constants

You can create a read-only named constant with the const keyword. The constant name must start with a letter, underscore, or dollar sign and can contain alphabetic, numeric, or underscore characters:

```javascript
const area_code = '515';
```

A constant cannot change the value through assignment or be redeclared, and it has to be initialized to a value.
JavaScript supports the standard variations of types:

- Number
- String
- Boolean
- Symbol (new in ECMAScript 6)
- Object:
  - Function
  - Array
  - Date
  - RegExp
- Null
- Undefined

**Number**

The **Number** type can represent both 32-bit integer and 64-bit floating point values. For example, the following line of code declares a variable to hold an integer value, which is defined by the literal 555:

```
var aNumber = 555;
```

To define a floating point value, you need to include a decimal point and one digit after the decimal point:

```
var aFloat = 555.0;
```

Essentially, there's no such thing as an integer in JavaScript. JavaScript uses a 64-bit floating point representation, which is the same as Java's double.

Hence, you would see something as follows:

```
EN-VedA:~$ node
> 0.1+0.2
0.30000000000000004
> (0.1+0.2)==0.3
false
```
I recommend that you read the exhaustive answer on Stack Overflow (http://stackoverflow.com/questions/588004/is-floating-point-math-broken) and (http://floating-point-gui.de/), which explains why this is the case. However, it is important to understand that floating point arithmetic should be handled with due care. In most cases, you will not have to rely on extreme precision of decimal points but if you have to, you can try using libraries such as big.js (https://github.com/MikeMcl/big.js) that try to solve this problem.

If you intend to code extremely precise financial systems, you should represent $ values as cents to avoid rounding errors. One of the systems that I worked on used to round off the Value Added Tax (VAT) amount to two decimal points. With thousands of orders a day, this rounding off amount per order became a massive accounting headache. We needed to overhaul the entire Java web service stack and JavaScript frontend for this.

A few special values are also defined as part of the Number type. The first two are Number.MAX_VALUE and Number.MIN_VALUE, which define the outer bounds of the Number value set. All ECMAScript numbers must fall between these two values, without exception. A calculation can, however, result in a number that does not fall in between these two numbers. When a calculation results in a number greater than Number.MAX_VALUE, it is assigned a value of Number.POSITIVE_INFINITY, meaning that it has no numeric value anymore. Likewise, a calculation that results in a number less than Number.MIN_VALUE is assigned a value of Number.NEGATIVE_INFINITY, which also has no numeric value. If a calculation returns an infinite value, the result cannot be used in any further calculations. You can use the isInfinite() method to verify if the calculation result is an infinity.

Another peculiarity of JavaScript is a special value called NaN (short for Not a Number). In general, this occurs when conversion from another type (String, Boolean, and so on) fails. Observe the following peculiarity of NaN:

```
> isNaN(NaN);  // true
> NaN==NaN;    // false
> isNaN("elephant");  // true
> NaN+5;  // NaN
```
The second line is strange—NaN is not equal to NaN. If NaN is part of any mathematical operation, the result also becomes NaN. As a general rule, stay away from using NaN in any expression. For any advanced mathematical operations, you can use the Math global object and its methods:

```javascript
> Math.E
2.718281828459045
> Math.SQRT2
1.4142135623730951
> Math.abs(-900)
900
> Math.pow(2,3)
8
```

You can use the parseInt() and parseFloat() methods to convert a string expression to an integer or float:

```javascript
> parseInt("230",10);
230
> parseInt("010",10);
10
> parseInt("010",8); //octal base
8
> parseInt("010",2); //binary
2
> + "4"
4
```

With parseInt(), you should provide an explicit base to prevent nasty surprises on older browsers. The last trick is just using a + sign to auto-convert the "42" string to a number, 42. It is also prudent to handle the parseInt() result with isNaN(). Let's see the following example:

```javascript
var underterminedValue = "elephant";
if (isNaN(parseInt(underterminedValue,2))){
  console.log("handle not a number case");
}
else{
  console.log("handle number case");
}
```
In this example, you are not sure of the type of the value that the `underterminedValue` variable can hold if the value is being set from an external interface. If `isNaN()` is not handled, `parseInt()` will cause an exception and the program can crash.

**String**

In JavaScript, strings are a sequence of Unicode characters (each character takes 16 bits). Each character in the string can be accessed by its index. The first character index is zero. Strings are enclosed inside " or ' – both are valid ways to represent strings. Let's see the following:

```
> console.log("Hippopotamus chewing gum");
Hippopotamus chewing gum
> console.log('Single quoted hippopotamus');
Single quoted hippopotamus
> console.log("Broken \n lines");
Broken lines
```

The last line shows you how certain character literals when escaped with a backslash \ can be used as special characters. The following is a list of such special characters:

- \n: Newline
- \t: Tab
- \b: Backspace
- \r: Carriage return
- \\: Backslash
- \': Single quote
- \": Double quote

You get default support for special characters and Unicode literals with JavaScript strings:

```
> '\xA9'
©
> '\u00A9'
©
```
One important thing about JavaScript Strings, Numbers, and Booleans is that they actually have wrapper objects around their primitive equivalent. The following example shows the usage of the wrapper objects:

```javascript
var s = new String("dummy"); //Creates a String object
console.log(s); //"dummy"
console.log(typeof s); //"object"
var nonObject = "1" + "2"; //Create a String primitive
console.log(typeof nonObject); //"string"
var objString = new String("1" + "2"); //Creates a String object
console.log(typeof objString); //"object"
```

//Helper functions
```javascript
console.log("Hello".length); //5
console.log("Hello".charAt(0)); //"H"
console.log("Hello".charAt(1)); //"e"
console.log("Hello".indexOf("e")); //1
console.log("Hello".lastIndexOf("l")); //3
console.log("Hello".startsWith("H")); //true
console.log("Hello".endsWith("o")); //true
```

//Helper functions
```javascript
console.log("Hello".length); //5
console.log("Hello".charAt(0)); //"H"
console.log("Hello".charAt(1)); //"e"
console.log("Hello".indexOf("e")); //1
console.log("Hello".lastIndexOf("l")); //3
console.log("Hello".startsWith("H")); //true
console.log("Hello".endsWith("o")); //true
```

//Helper functions
```javascript
var splitStringByWords = "Hello World".split(" ");
console.log(splitStringByWords); //["Hello", "World"]
var splitStringByChars = "Hello World".split(" ");
console.log(splitStringByChars); //["H", "e", "l", "o", " ", ",", "W", "o", "r", "l", "d"]
```

This kind of string is also known as a template string and can be used for string interpolation. JavaScript allows Python-like string interpolation using this syntax.

JavaScript allows multiline strings also. Strings enclosed within ` (Grave accent—https://en.wikipedia.org/wiki/Grave_accent) are considered multiline. Let's see the following example:

```javascript
> console.log(`string text on first line
string text on second line `);
"string text on first line
string text on second line 
```

This kind of string is also known as a template string and can be used for string interpolation. JavaScript allows Python-like string interpolation using this syntax.
Normally, you would do something similar to the following:

```javascript
var a=1, b=2;
console.log("Sum of values is :" + (a+b) + " and multiplication is :
 + (a*b));
```

However, with string interpolation, things become a bit clearer:

```javascript
console.log(`Sum of values is :${a+b} and multiplication is :
 ${a*b}`);
```

**Undefined values**

JavaScript indicates an absence of meaningful values by two special values—null, when the non-value is deliberate, and undefined, when the value is not assigned to the variable yet. Let's see the following example:

```javascript
> var xl;
> console.log(typeof xl);
undefined
> console.log(null==undefined);
true
```

**Booleans**

JavaScript Boolean primitives are represented by `true` and `false` keywords. The following rules govern what becomes false and what turns out to be true:

- False, 0, the empty string (`""`), NaN, null, and undefined are represented as false
- Everything else is true

JavaScript Booleans are tricky primarily because the behavior is radically different in the way you create them.

There are two ways in which you can create Booleans in JavaScript:

- You can create primitive Booleans by assigning a true or false literal to a variable. Consider the following example:
  ```javascript
  var pBooleanTrue = true;
  var pBooleanFalse = false;
  ```
• Use the `Boolean()` function; this is an ordinary function that returns a primitive Boolean:

```javascript
var fBooleanTrue = Boolean(true);
var fBooleanFalse = Boolean(false);
```

Both these methods return expected truthy or falsy values. However, if you create a Boolean object using the `new` operator, things can go really wrong.

Essentially, when you use the `new` operator and the `Boolean(value)` constructor, you don't get a primitive `true` or `false` in return, you get an object instead—and unfortunately, JavaScript considers an object as truthy:

```javascript
var oBooleanTrue = new Boolean(true);
var oBooleanFalse = new Boolean(false);
console.log(oBooleanTrue); //true
console.log(typeof oBooleanTrue); //object
if(oBooleanFalse){
  console.log("I am seriously truthy, don't believe me");
}
>"I am seriously truthy, don't believe me"

if(oBooleanTrue){
  console.log("I am also truthy, see ?");
}
>"I am also truthy, see ?"

//Use `valueOf()` to extract real value within the Boolean object
if(oBooleanFalse.valueOf()){  
  console.log("With valueOf, I am false");
}else{
  console.log("Without valueOf, I am still truthy");
}
>"Without valueOf, I am still truthy"
```

So, the smart thing to do is to always avoid Boolean constructors to create a new Boolean object. It breaks the fundamental contract of Boolean logic and you should stay away from such difficult-to-debug buggy code.
Chapter 1

The instanceof operator

One of the problems with using reference types to store values has been the use of the `typeof` operator, which returns `object` no matter what type of object is being referenced. To provide a solution, you can use the `instanceof` operator. Let's see some examples:

```javascript
var aStringObject = new String("string");
console.log(typeof aStringObject);        //"object"
console.log(aStringObject instanceof String);    //true
var aString = "This is a string";
console.log(aString instanceof String);     //false
```

The third line returns `false`. We will discuss why this is the case when we discuss prototype chains.

Date objects

JavaScript does not have a date data type. Instead, you can use the `Date` object and its methods to work with dates and times in your applications. A Date object is pretty exhaustive and contains several methods to handle most date- and time-related use cases.

JavaScript treats dates similarly to Java. JavaScript store dates as the number of milliseconds since January 1, 1970, 00:00:00.

You can create a Date object using the following declaration:

```javascript
var dataObject = new Date([parameters]);
```

The parameters for the Date object constructors can be as follows:

- No parameters creates today's date and time. For example, `var today = new Date();`.
- A String representing a date as Month day, year hours:minutes:seconds. For example, `var twoThousandFifteen = new Date("December 31, 2015 23:59:59");`. If you omit hours, minutes, or seconds, the value will be set to 0.
- A set of integer values for the year, month, and day. For example, `var christmas = new Date(2015, 11, 25);`.
- A set of integer values for the year, month, day, hour, minute, and seconds. For example, `var christmas = new Date(2015, 11, 25, 21, 00, 0);`.
Here are some examples on how to create and manipulate dates in JavaScript:

```javascript
var today = new Date();
console.log(today.getDate()); // 27
console.log(today.getMonth()); // 4
console.log(today.getFullYear()); // 2015
console.log(today.getHours()); // 23
console.log(today.getMinutes()); // 13
console.log(today.getSeconds()); // 10
// number of milliseconds since January 1, 1970, 00:00:00 UTC
console.log(today.getTime()); // 1432748611392
console.log(today.getTimezoneOffset()); // -330 Minutes
```

// Calculating elapsed time
var start = Date.now();
// loop for a long time
for (var i=0; i<100000; i++);
var end = Date.now();
var elapsed = end - start; // elapsed time in milliseconds
console.log(elapsed); // 71

For any serious applications that require fine-grained control over date and time objects, we recommend using libraries such as Moment.js (https://github.com/moment/moment), Timezone.js (https://github.com/mde/timezone-js), or date.js (https://github.com/MatthewMueller/date). These libraries simplify a lot of recurrent tasks for you and help you focus on other important things.

### The + operator

The `+` operator, when used as a unary, does not have any effect on a number. However, when applied to a String, the `+` operator converts it to numbers as follows:

```javascript
var a=25;
a+=a; // No impact on a's value
console.log(a); // 25

var b="70";
cconsole.log(typeof b); // string
b+=b; // converts string to number
console.log(b); // 70
console.log(typeof b); // number
```
The + operator is used often by a programmer to quickly convert a numeric representation of a String to a number. However, if the String literal is not something that can be converted to a number, you get slightly unpredictable results as follows:

```javascript
var c="foo*;
c=+c;            //Converts foo to number
console.log(c);  //NaN
console.log(typeof c);  //number

var zero="";
zero=+zero; //empty strings are converted to 0
console.log(zero);
console.log(typeof zero);
```

We will discuss the effects of the + operator on several other data types later in the text.

**The ++ and -- operators**

The ++ operator is a shorthand version of adding 1 to a value and -- is a shorthand to subtract 1 from a value. Java and C have equivalent operators and most will be familiar with them. How about this?

```javascript
var a= 1;
var b= a++;  
c = b = a++;
console.log(a);  //2
console.log(b);  //1
```

Err, what happened here? Shouldn't the b variable have the value 2? The ++ and -- operators are unary operators that can be used either prefix or postfix. The order in which they are used matters. When ++ is used in the prefix position as ++a, it increments the value before the value is returned from the expression rather than after as with a++. Let's see the following code:

```javascript
var a= 1;
var b= ++a;
console.log(a);  //2
console.log(b);  //2
```

Many programmers use the chained assignments to assign a single value to multiple variables as follows:

```javascript
var a, b, c;
a = b = c = 0;
```
This is fine because the assignment operator (=) results in the value being assigned. In this case, \( c=0 \) is evaluated to \( 0 \); this would result in \( b=0 \), which also evaluates to \( 0 \), and hence, \( a=0 \) is evaluated.

However, a slight change to the previous example yields extraordinary results. Consider this:

```javascript
var a = b = 0;
```

In this case, only the \( a \) variable is declared with `var`, while the \( b \) variable is created as an accidental global. (If you are in the strict mode, you will get an error for this.) With JavaScript, be careful what you wish for, you might get it.

**Boolean operators**

There are three Boolean operators in JavaScript—\( \text{AND}(\&), \text{OR}(|) \), and \( \text{NOT}(!) \).

Before we discuss logical AND and OR operators, we need to understand how they produce a Boolean result. Logical operators are evaluated from left to right and they are tested using the following short-circuit rules:

- **Logical AND**: If the first operand determines the result, the second operand is not evaluated.

  In the following example, I have highlighted the right-hand side expression if it gets executed as part of short-circuit evaluation rules:

  ```javascript
  console.log(true && true); // true AND true returns true
  console.log(true && false); // true AND false returns false
  console.log(false && true); // false AND true returns false
  console.log("Foo" && "Bar"); // Foo(true) AND Bar(true) returns Bar
  console.log(false && "Foo"); // false && Foo(true) returns false
  console.log("Foo" && false); // Foo(true) && false returns false
  console.log(false && (1 == 2)); // false && false(1==2) returns false
  ```
• **Logical OR**: If the first operand is true, the second operand is not evaluated:

```javascript
console.log(true || true); // true AND true returns true
console.log(true || false); // true AND false returns true
console.log(false || true); // false AND true returns true
console.log("Foo" || "Bar"); // Foo(true) AND Bar(true) returns Foo
console.log(false || "Foo"); // false && Foo(true) returns Foo
console.log("Foo" || false); // Foo(true) && false returns Foo
console.log(false || (1 == 2)); // false && false(1==2) returns false
```

However, both logical AND and logical OR can also be used for non-Boolean operands. When either the left or right operand is not a primitive Boolean value, AND and OR do not return Boolean values.

Now we will explain the three logical Boolean operators:

• **Logical AND(&&)**: If the first operand object is *falsy*, it returns that object. If its *truthy*, the second operand object is returned:

```javascript
console.log(0 && "Foo");  //First operand is falsy - return it
console.log("Foo" && "Bar"); //First operand is truthy, return the second operand
```

• **Logical OR(||)**: If the first operand is *truthy*, it's returned. Otherwise, the second operand is returned:

```javascript
console.log(0 || "Foo");  //First operand is falsy - return second operand
console.log("Foo" || "Bar"); //First operand is truthy, return it
console.log(0 || false); //First operand is falsy, return second operand
```

The typical use of a logical OR is to assign a default value to a variable:

```javascript
function greeting(name){
    name = name || "John";
    console.log("Hello " + name);
}

greeting("Johnson"); // alerts "Hi Johnson"
greeting(); //alerts "Hello John"
```

You will see this pattern frequently in most professional JavaScript libraries. You should understand how the defaulting is done by using a logical OR operator.
• **Logical NOT**: This always returns a Boolean value. The value returned depends on the following:

//If the operand is an object, false is returned.
var s = new String("string");
console.log(!s);  //false

//If the operand is the number 0, true is returned.
var t = 0;
console.log(!t);  //true

//If the operand is any number other than 0, false is returned.
var x = 11;
console.log(!x);  //false

//If operand is null or NaN, true is returned
var y =null;
var z = NaN;
console.log(!y);  //true
console.log(!z);  //true

//If operand is undefined, you get true
var foo;
console.log(!foo);  //true

Additionally, JavaScript supports C-like ternary operators as follows:

```javascript
var allowedToDrive = (age > 21) ? "yes" : "no";
```

If `(age > 21)`, the expression after `?` will be assigned to the `allowedToDrive` variable and the expression after `:` is assigned otherwise. This is equivalent to an if-else conditional statement. Let's see another example:

```javascript
function isAllowedToDrive(age){
  if(age>21){
    return true;
  }else{
    return false;
  }  
}
console.log(isAllowedToDrive(22));
```
In this example, the `isAllowedToDrive()` function accepts one integer parameter, `age`. Based on the value of this variable, we return true or false to the calling function. This is a well-known and most familiar if-else conditional logic. Most of the time, if-else keeps the code easier to read. For simpler cases of single conditions, using the ternary operator is also okay, but if you see that you are using the ternary operator for more complicated expressions, try to stick with if-else because it is easier to interpret if-else conditions than a very complex ternary expression.

If-else conditional statements can be nested as follows:

```javascript
if (condition1) {
    statement1
} else if (condition2) {
    statement2
} else if (condition3) {
    statement3
}
...
} else {
    statementN
}
```

Purely as a matter of taste, you can indent the nested `else if` as follows:

```javascript
if (condition1) {
    statement1
} else
    if (condition2) {
```

Do not use assignments in place of a conditional statement. Most of the time, they are used because of a mistake as follows:

```javascript
if(a=b) {
    //do something
}
```

Mostly, this happens by mistake; the intended code was `if(a==b)`, or better, `if(a===b)`. When you make this mistake and replace a conditional statement with an assignment statement, you end up committing a very difficult-to-find bug. However, if you really want to use an assignment statement with an if statement, make sure that you make your intentions very clear.
One way is to put extra parentheses around your assignment statement:

```javascript
if((a=b)){
    //this is really something you want to do
}
```

Another way to handle conditional execution is to use switch-case statements. The switch-case construct in JavaScript is similar to that in C or Java. Let’s see the following example:

```javascript
function sayDay(day){
    switch(day){
        case 1: console.log("Sunday");
            break;
        case 2: console.log("Monday");
            break;
        default:
            console.log("We live in a binary world. Go to Pluto");
    }
}

sayDay(1); //Sunday
sayDay(3); //We live in a binary world. Go to Pluto
```

One problem with this structure is that you have break out of every case; otherwise, the execution will fall through to the next level. If we remove the break statement from the first case statement, the output will be as follows:

```javascript
>sayDay(1);
Sunday
Monday
```

As you can see, if we omit the break statement to break the execution immediately after a condition is satisfied, the execution sequence follows to the next level. This can lead to difficult-to-detect problems in your code. However, this is also a popular style of writing conditional logic if you intend to fall through to the next level:

```javascript
function debug(level,msg){
    switch(level){
        case "INFO": //intentional fall-through
        case "WARN" :
        case "DEBUG": console.log(level + ": " + msg);
            break;
        case "ERROR": console.error(msg);
    }
}
```
debug("INFO","Info Message");
debug("DEBUG","Debug Message");
debug("ERROR","Fatal Exception");

In this example, we are intentionally letting the execution fall through to write a concise switch-case. If levels are either INFO, WARN, or DEBUG, we use the switch-case to fall through to a single point of execution. We omit the break statement for this. If you want to follow this pattern of writing switch statements, make sure that you document your usage for better readability.

Switch statements can have a default case to handle any value that cannot be evaluated by any other case.

JavaScript has a while and do-while loop. The while loop lets you iterate a set of expressions till a condition is met. The following first example iterates the statements enclosed within {} till the \( i < 10 \) expression is true. Remember that if the value of the \( i \) counter is already greater than 10, the loop will not execute at all:

```javascript
var i=0;
while(i<10){
i=i+1;
    console.log(i); 
}
```

The following loop keeps executing till infinity because the condition is always true – this can lead to disastrous effects. Your program can use up all your memory or something equally unpleasant:

```javascript
//infinite loop
while(true){
    //keep doing this 
}
```

If you want to make sure that you execute the loop at least once, you can use the do-while loop (sometimes known as a post-condition loop):

```javascript
var choice;
do {
    choice=getChoiceFromUserInput();
} while(!isInputValid(input));
```
In this example, we are asking the user for an input till we find a valid input from the user. While the user types invalid input, we keep asking for an input to the user. It is always argued that, logically, every do-while loop can be transformed into a while loop. However, a do-while loop has a very valid use case like the one we just saw where you want the condition to be checked only after there has been one execution of the loop block.

JavaScript has a very powerful loop similar to C or Java—the for loop. The for loop is popular because it allows you to define the control conditions of the loop in a single line.

The following example prints Hello five times:

```javascript
for (var i=0; i<5; i++){
  console.log("Hello");
}
```

Within the definition of the loop, you defined the initial value of the loop counter i to be 0, you defined the i<5 exit condition, and finally, you defined the increment factor.

All three expressions in the previous example are optional. You can omit them if required. For example, the following variations are all going to produce the same result as the previous loop:

```javascript
var x=0;
//Omit initialization
for (;x<5;x++){  
  console.log("Hello");
}

//Omit exit condition
for (var j=0;j++){
  //exit condition
  if(j>=5){
    break;
  }else{
    console.log("Hello");
  }
}

//Omit increment
for (var k=0; k<5;){
  console.log("Hello");
  k++;
}
```
You can also omit all three of these expressions and write for loops. One interesting idiom used frequently is to use for loops with empty statements. The following loop is used to set all the elements of the array to 100. Notice how there is no body to the for-loop:

```javascript
var arr = [10, 20, 30];
// Assign all array values to 100
for (i = 0; i < arr.length; arr[i++] = 100);
console.log(arr);
```

The empty statement here is just the single that we see after the for loop statement. The increment factor also modifies the array content. We will discuss arrays later in the book, but here it's sufficient to see that the array elements are set to the 100 value within the loop definition itself.

### Equality

JavaScript offers two modes of equality—strict and loose. Essentially, loose equality will perform the type conversion when comparing two values, while strict equality will check the values without any type conversion. A strict equality check is performed by `===` while a loose equality check is performed by `==`.

ECMAScript 6 also offers the `Object.is` method to do a strict equality check like `===`. However, `Object.is` has a special handling for NaN: -0 and +0. When `NaN===NaN` and `NaN==NaN` evaluates to false, `Object.is(NaN, NaN)` will return true.

#### Strict equality using `===`

Strict equality compares two values without any implicit type conversions. The following rules apply:

- If the values are of a different type, they are unequal.
- For non-numerical values of the same type, they are equal if their values are the same.
- For primitive numbers, strict equality works for values. If the values are the same, `===` results in `true`. However, a `NaN` doesn't equal to any number and `NaN===<a number>` would be a `false`.
Strict equality is always the correct equality check to use. Make it a rule to always use `===` instead of `==`:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;&quot; === &quot;0&quot;</td>
<td>false</td>
</tr>
<tr>
<td>0 === &quot;&quot;</td>
<td>false</td>
</tr>
<tr>
<td>0 === &quot;0&quot;</td>
<td>false</td>
</tr>
<tr>
<td>false === &quot;false&quot;</td>
<td>false</td>
</tr>
<tr>
<td>false === &quot;0&quot;</td>
<td>false</td>
</tr>
<tr>
<td>false === undefined</td>
<td>false</td>
</tr>
<tr>
<td>false === null</td>
<td>false</td>
</tr>
<tr>
<td>null === undefined</td>
<td>false</td>
</tr>
</tbody>
</table>

In case of comparing objects, we get results as follows:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>{} === {}</td>
<td>false</td>
</tr>
<tr>
<td>new String('bah') === 'bah';</td>
<td>false</td>
</tr>
<tr>
<td>new Number(1) === 1;</td>
<td>false</td>
</tr>
<tr>
<td>var bar = {};</td>
<td>true</td>
</tr>
<tr>
<td>bar === bar;</td>
<td>true</td>
</tr>
</tbody>
</table>

The following are further examples that you should try on either JS Bin or Node REPL:

```javascript
var n = 0;
var o = new String("0");
var s = "0";
var b = false;

console.log(n === n); // true - same values for numbers
console.log(o === o); // true - non numbers are compared for their values
console.log(s === s); // true - ditto

console.log(n === o); // false - no implicit type conversion, types are different
console.log(n === s); // false - types are different
console.log(o === s); // false - types are different
console.log(null === undefined); // false
console.log(o === null); // false
console.log(o === undefined); // false
```

You can use `!==` to handle the **Not Equal To** case while doing strict equality checks.
Weak equality using ==

Nothing should tempt you to use this form of equality. Seriously, stay away from this form. There are many bad things with this form of equality primarily due to the weak typing in JavaScript. The equality operator, ==, first tries to coerce the type before doing a comparison. The following examples show you how this works:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;&quot; == &quot;0&quot;</td>
<td>false</td>
</tr>
<tr>
<td>0 == &quot;&quot;</td>
<td>true</td>
</tr>
<tr>
<td>0 == &quot;0&quot;</td>
<td>true</td>
</tr>
<tr>
<td>false == &quot;false&quot;</td>
<td>false</td>
</tr>
<tr>
<td>false == &quot;0&quot;</td>
<td>true</td>
</tr>
<tr>
<td>false == undefined</td>
<td>false</td>
</tr>
<tr>
<td>false == null</td>
<td>false</td>
</tr>
<tr>
<td>null == undefined</td>
<td>true</td>
</tr>
</tbody>
</table>

From these examples, it’s evident that weak equality can result in unexpected outcomes. Also, implicit type coercion is costly in terms of performance. So, in general, stay away from weak equality in JavaScript.

JavaScript types

We briefly discussed that JavaScript is a dynamic language. If you have a previous experience of strongly typed languages such as Java, you may feel a bit uncomfortable about the complete lack of type checks that you are used to. Purists argue that JavaScript should claim to have tags or perhaps subtypes, but not types. Though JavaScript does not have the traditional definition of types, it is absolutely essential to understand how JavaScript handles data types and coercion internally. Every nontrivial JavaScript program will need to handle value coercion in some form, so it’s important that you understand the concept well.

Explicit coercion happens when you modify the type yourself. In the following example, you will convert a number to a String using the toString() method and extract the second character out of it:

```javascript
var fortyTwo = 42;
console.log(fortyTwo.toString()[1]); // prints "2"
```

This is an example of an explicit type conversion. Again, we are using the word type loosely because type was not enforced anywhere when you declared the fortyTwo variable.
However, there are many different ways in which such coercion can happen. Coercion happening explicitly can be easy to understand and mostly reliable; but if you’re not careful, coercion can happen in very strange and surprising ways.

Confusion around coercion is perhaps one of the most talked about frustrations for JavaScript developers. To make sure that you never have this confusion in your mind, let's revisit types in JavaScript. We talked about some concepts earlier:

```javascript
typeof 1             === "number";    // true
typeof "1"           === "string";    // true
typeof { age: 39 }   === "object";    // true
typeof Symbol()      === "symbol";    // true
typeof undefined     === "undefined"; // true
typeof true          === "boolean";   // true
```

So far, so good. We already knew this and the examples that we just saw reinforce our ideas about types.

Conversion of a value from one type to another is called casting or explicit coercion. JavaScript also does implicit coercion by changing the type of a value based on certain guesses. These guesses make JavaScript work around several cases and unfortunately make it fail quietly and unexpectedly. The following snippet shows cases of explicit and implicit coercion:

```javascript
var t=1;
var u=""+t; //implicit coercion
console.log(typeof t);  //"number"
console.log(typeof u);  //"string"
var v=String(t);  //Explicit coercion
console.log(typeof v);  //"string"
var x=null
console.log(""+x);  //"null"
```

It is easy to see what is happening here. When you use ""+t to a numeric value of t (1, in this case), JavaScript figures out that you are trying to concatenate something with a "" string. As only strings can be concatenated with other strings, JavaScript goes ahead and converts a numeric 1 to a "1" string and concatenates both into a resulting string value. This is what happens when JavaScript is asked to convert values implicitly. However, String(t) is a very deliberate call to convert a number to a String. This is an explicit conversion of types. The last bit is surprising. We are concatenating null with ""—shouldn't this fail?

So how does JavaScript do type conversions? How will an abstract value become a String or number or Boolean? JavaScript relies on toString(), toNumber(), and toBoolean() methods to do this internally.
When a non-String value is coerced into a String, JavaScript uses the `toString()` method internally to do this. All primitives have a natural string form—null has a string form of "null", undefined has a string form of "undefined", and so on. For Java developers, this is analogous to a class having a `toString()` method that returns a string representation of the class. We will see exactly how this works in case of objects.

So essentially you can do something similar to the following:

```javascript
var a="abc";
console.log(a.length);
console.log(a.toUpperCase());
```

If you are keenly following and typing all these little snippets, you would have realized something strange in the previous snippet. How are we calling properties and methods on primitives? How come primitives have objects such as properties and methods? They don't.

As we discussed earlier, JavaScript kindly wraps these primitives in their wrappers by default thus making it possible for us to directly access the wrapper’s methods and properties as if they were of the primitives themselves.

When any non-number value needs to be coerced into a number, JavaScript uses the `toNumber()` method internally: true becomes 1, undefined becomes NaN, false becomes 0, and null becomes 0. The `toNumber()` method on strings works with literal conversion and if this fails, the method returns NaN.

What about some other cases?

```javascript
typeof null ==="object" //true
```

Well, null is an object? Yes, an especially long-lasting bug makes this possible. Due to this bug, you need to be careful while testing if a value is null:

```javascript
var x = null;
if (!x && typeof x === "object"){
    console.log("100% null");
}
```

What about other things that may have types, such as functions?

```javascript
f = function test() {
    return 12;
}
console.log(typeof f === "function"); //prints "true"
```
What about arrays?

    console.log (typeof [1,2,3,4]); //"object"

Sure enough, they are also objects. We will take a detailed look at functions and arrays later in the book.

In JavaScript, values have types, variables don't. Due to the dynamic nature of the language, variables can hold any value at any time.

JavaScript doesn't does not enforce types, which means that the language doesn't insist that a variable always hold values of the same initial type that it starts out with. A variable can hold a String, and in the next assignment, hold a number, and so on:

```javascript
var a = 1;
typeof a; // "number"
a = false;
typeof a; // "boolean"
```

The `typeof` operator always returns a String:

```javascript
typeof typeof 1; // "string"
```

**Automatic semicolon insertion**

Although JavaScript is based on the C style syntax, it does not enforce the use of semicolons in the source code.

However, JavaScript is not a semicolon-less language. A JavaScript language parser needs the semicolons in order to understand the source code. Therefore, the JavaScript parser automatically inserts them whenever it encounters a parse error due to a missing semicolon. It's important to note that **automatic semicolon insertion (ASI)** will only take effect in the presence of a newline (also known as a line break). Semicolons are not inserted in the middle of a line.

Basically, if the JavaScript parser parses a line where a parser error would occur (a missing expected ;) and it can insert one, it does so. What are the criteria to insert a semicolon? Only if there's nothing but white space and/or comments between the end of some statement and that line's newline/line break.

There have been raging debates on ASI—a feature justifiably considered to be a very bad design choice. There have been epic discussions on the Internet, such as https://github.com/twbs/bootstrap/issues/3057 and https://brendaneich.com/2012/04/the-infernal-semicolon/.
Before you judge the validity of these arguments, you need to understand what is affected by ASI. The following statements are affected by ASI:

- An empty statement
- A var statement
- An expression statement
- A do-while statement
- A continue statement
- A break statement
- A return statement
- A throw statement

The idea behind ASI is to make semicolons optional at the end of a line. This way, ASI helps the parser to determine when a statement ends. Normally, it ends with a semicolon. ASI dictates that a statement also ends in the following cases:

- A line terminator (for example, a newline) is followed by an illegal token
- A closing brace is encountered
- The end of the file has been reached

Let's see the following example:

```javascript
if (a < 1) a = 1 console.log(a)
```

The `console` token is illegal after `1` and triggers ASI as follows:

```javascript
if (a < 1) a = 1; console.log(a);
```

In the following code, the statement inside the braces is not terminated by a semicolon:

```javascript
function add(a,b) { return a+b }
```

ASI creates a syntactically correct version of the preceding code:

```javascript
function add(a,b) { return a+b; }
```
JavaScript style guide

Every programming language develops its own style and structure. Unfortunately, new developers don't put much effort in learning the stylistic nuances of a language. It is very difficult to develop this skill later once you have acquired bad practices. To produce beautiful, readable, and easily maintainable code, it is important to learn the correct style. There are a ton of style suggestions. We will be picking the most practical ones. Whenever applicable, we will discuss the appropriate style. Let's set some stylistic ground rules.

Whitespaces

Though whitespace is not important in JavaScript, the correct use of whitespace can make the code easy to read. The following guidelines will help in managing whitespaces in your code:

- Never mix spaces and tabs.
- Before you write any code, choose between soft indents (spaces) or real tabs. For readability, I always recommend that you set your editor's indent size to two characters—which means two spaces or two spaces representing a real tab.
- Always work with the show invisibles setting turned on. The benefits of this practice are as follows:
  - Enforced consistency.
  - Eliminates the end-of-line white spaces.
  - Eliminates blank line white spaces.
  - Commits and diffs that are easier to read.
  - Uses EditorConfig (http://editorconfig.org/) when possible.

Parentheses, line breaks, and braces

If, else, for, while, and try always have spaces and braces and span multiple lines. This style encourages readability. Let's see the following code:

```javascript
// Cramped style (Bad)
if (condition) doSomeTask();
while (condition) i++;
for (var i = 0; i < 10; i++) iterate();

// Use whitespace for better readability (Good)
```

```javascript
if (condition) {
  doSomeTask();
}
while (condition) {
  i++;
}
for (var i = 0; i < 10; i++) {
  iterate();
}
```
// Place 1 space before the leading brace.
if (condition) {
// statements
}

while (condition) {
// statements
}

for (var i = 0; i < 100; i++) {
// statements
}

// Better:

var i,
    length = 100;

for (i = 0; i < length; i++) {
// statements
}

// Or...

var i = 0,
    length = 100;

for (; i < length; i++) {
// statements
}

var value;

for (value in object) {
// statements
}

if (true) {
// statements
} else {
// statements
}
JavaScript Primer

// Set off operators with spaces.
// bad
var x = y + 5;

// good
var x = y + 5;

// End files with a single newline character.
// bad
(function(global) {
  // ...stuff...
})(this);

// bad
(function(global) {
  // ...stuff...
})(this);

// good
(function(global) {
  // ...stuff...
})(this);

Quotes
Whether you prefer single or double quotes shouldn't matter; there is no difference in how JavaScript parses them. However, for the sake of consistency, never mix quotes in the same project. Pick one style and stick with it.

End of lines and empty lines
Whitespace can make it impossible to decipher code diffs and changelists. Many editors allow you to automatically remove extra empty lines and end of lines—you should use these.
**Type checking**

Checking the type of a variable can be done as follows:

```javascript
//String:
typeOf variable === "string"
//Number:
typeOf variable === "number"
//Boolean:
typeOf variable === "boolean"
//Object:
typeOf variable === "object"
//null:
variable === null
//null or undefined:
variable == null
```

**Type casting**

Perform type coercion at the beginning of the statement as follows:

```javascript
// bad
const totalScore = this.reviewScore + '';
// good
const totalScore = String(this.reviewScore);
```

Use `parseInt()` for Numbers and always with a radix for the type casting:

```javascript
const inputValue = '4';
// bad
const val = new Number(inputValue);
// bad
const val = +inputValue;
// bad
const val = inputValue >> 0;
// bad
const val = parseInt(inputValue);
// good
const val = Number(inputValue);
// good
const val = parseInt(inputValue, 10);
```
The following example shows you how to type cast using Booleans:

```javascript
const age = 0; // bad
const hasAge = new Boolean(age); // good
const hasAge = Boolean(age); // good
const hasAge = !!age;
```

**Conditional evaluation**

There are various stylistic guidelines around conditional statements. Let's study the following code:

```javascript
// When evaluating that array has length,
// WRONG:
if ( array.length > 0 ) ...

// evaluate truthiness (GOOD):
if ( array.length ) ...

// When evaluating that an array is empty,
// (BAD):
if ( array.length === 0 ) ...

// evaluate truthiness (GOOD):
if ( !array.length ) ...

// When checking if string is not empty,
// (BAD):
if ( string !== "" ) ...

// evaluate truthiness (GOOD):
if ( string ) ...

// When checking if a string is empty,
// BAD:
if ( string === "" ) ...

// evaluate falsy-ness (GOOD):
if ( !string ) ...

// When checking if a reference is true,
// BAD:
if ( foo === true ) ...
```
// GOOD
if ( foo ) ...

// When checking if a reference is false,
// BAD:
if ( foo === false ) ...

// GOOD
if ( !foo ) ...

// this will also match: 0, "", null, undefined, NaN
// If you MUST test for a boolean false, then use
if ( foo === false ) ...

// a reference that might be null or undefined, but NOT false, "" or 0,
// BAD:
if ( foo === null || foo === undefined ) ...

// GOOD
if ( foo == null ) ...

// Don't complicate matters
return x === 0 ? 'sunday' : x === 1 ? 'Monday' : 'Tuesday';

// Better:
if (x === 0) {
    return 'Sunday';
} else if (x === 1) {
    return 'Monday';
} else {
    return 'Tuesday';
}

// Even Better:
switch (x) {
    case 0:
        return 'Sunday';
    case 1:
        return 'Monday';
    default:
        return 'Tuesday';
}
Naming

Naming is super important. I am sure that you have encountered code with terse and undecipherable naming. Let's study the following lines of code:

```javascript
// Avoid single letter names. Be descriptive with your naming.
// bad
function q() {

}

// good
function query() {

}

// Use camelCase when naming objects, functions, and instances.
// bad
const OBJECT = {};
const this_is_object = {};
function c() {}

// good
const thisIsObject = {};
function thisIsFunction() {}

// Use PascalCase when naming constructors or classes.
// bad
function user(options) {
    this.name = options.name;
}

const bad = new user({
    name: 'nope',
});

// good
class User {
    constructor(options) {
        this.name = options.name;
    }
}
```
const good = new User({
  name: 'yup',
});

// Use a leading underscore _ when naming private properties.
// bad
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';

// good
this._firstName = 'Panda';

The eval() method is evil
The eval() method, which takes a String containing JavaScript code, compiles it and
runs it, is one of the most misused methods in JavaScript. There are a few situations
where you will find yourself using eval(), for example, when you are building an
expression based on the user input.

However, most of the time, eval() is used is just because it gets the job done. The
eval() method is too hacky and makes the code unpredictable. It's slow, unwieldy,
and tends to magnify the damage when you make a mistake. If you are considering
using eval(), then there is probably a better way.

The following snippet shows the usage of eval():

    console.log(typeof eval(new String("1+1")));  // "object"
    console.log(eval(new String("1+1")));          //1+1
    console.log(eval("1+1"));                      // 2
    console.log(typeof eval("1+1"));              // returns "number"
    var expression = new String("1+1");
    console.log(eval(expression.toString()));     //2

I will refrain from showing other uses of eval() and make sure that you are
discouraged enough to stay away from it.

The strict mode
ECMAScript 5 has a strict mode that results in cleaner JavaScript, with fewer
unsafe features, more warnings, and more logical behavior. The normal (non-strict)
mode is also called sloppy mode. The strict mode can help you avoid a few sloppy
programming practices. If you are starting a new JavaScript project, I would highly
recommend that you use the strict mode by default.
You switch on the strict mode by typing the following line first in your JavaScript file or in your `<script>` element:

```javascript
'use strict';
```

Note that JavaScript engines that don't support ECMAScript 5 will simply ignore the preceding statement and continue as non-strict mode.

If you want to switch on the strict mode per function, you can do it as follows:

```javascript
function foo() {
    'use strict';
}
```

This is handy when you are working with a legacy code base where switching on the strict mode everywhere may break things.

If you are working on an existing legacy code, be careful because using the strict mode can break things. There are caveats on this:

**Enabling the strict mode for an existing code can break it**

The code may rely on a feature that is not available anymore or on behavior that is different in a sloppy mode than in a strict mode. Don't forget that you have the option to add single strict mode functions to files that are in the sloppy mode.

**Package with care**

When you concatenate and/or minify files, you have to be careful that the strict mode isn't switched off where it should be switched on or vice versa. Both can break code.

The following sections explain the strict mode features in more detail. You normally don't need to know them as you will mostly get warnings for things that you shouldn't do anyway.

**Variables must be declared in strict mode**

All variables must be explicitly declared in strict mode. This helps to prevent typos. In the sloppy mode, assigning to an undeclared variable creates a global variable:

```javascript
function sloppyFunc() {
    sloppyVar = 123;
} sloppyFunc(); // creates global variable `sloppyVar`
console.log(sloppyVar); // 123
```
In the strict mode, assigning to an undeclared variable throws an exception:

```javascript
function strictFunc() {
  'use strict';
  strictVar = 123;
}
strictFunc();  // ReferenceError: strictVar is not defined
```

The eval() function is cleaner in strict mode
In strict mode, the eval() function becomes less quirky: variables declared in the evaluated string are not added to the scope surrounding eval() anymore.

Features that are blocked in strict mode
The with statement is not allowed. (We will discuss this in the book later.) You get a syntax error at compile time (when loading the code).

In the sloppy mode, an integer with a leading zero is interpreted as octal (base 8) as follows:

```
> 010 === 8
true
```

In strict mode, you get a syntax error if you use this kind of literal:

```javascript
function f() {
  'use strict';
  return 010
}
//SyntaxError: Octal literals are not allowed in
```

Running JSHint
JSHint is a program that flags suspicious usage in programs written in JavaScript. The core project consists of a library itself as well as a command line interface (CLI) program distributed as a Node module.

If you have Node.js installed, you can install JSHint using npm as follows:

```bash
npm install jshint -g
```
Once JSHint is installed, you can lint a single or multiple JavaScript files. Save the following JavaScript code snippet in the test.js file:

```javascript
function f(condition) {
    switch (condition) {
    case 1:
        console.log(1);
    case 2:
        console.log(1);
    }
}
```

When we run the file using JSHint, it will warn us of a missing `break` statement in the switch case as follows:

```
> jshint test.js

  test.js: line 4, col 19, Expected a 'break' statement before 'case'.

1 error
```

JSHint is configurable to suit your needs. Check the documentation at [http://jshint.com/docs/](http://jshint.com/docs/) to see how you can customize JSHint according to your project needs. I use JSHint extensively and suggest you start using it. You will be surprised to see how many hidden bugs and stylistic issues you will be able to fix in your code with such a simple tool.

You can run JSHint at the root of your project and lint the entire project. You can place JSHint directives in the `.jshintrc` file. This file may look something as follows:

```json
{
    "asi": false,
    "expr": true,
    "loopfunc": true,
    "curly": false,
    "evil": true,
    "white": true,
    "undef": true,
    "indent": 4
}
```
Summary

In this chapter, we set some foundations around JavaScript grammar, types, and stylistic considerations. We have consciously not talked about other important aspects such as functions, variable scopes, and closures primarily because they deserve their own place in this book. I am sure that this chapter helps you understand some of the primary concepts of JavaScript. With these foundations in place, we will take a look at how we can write professional quality JavaScript code.
Where to buy this book

You can buy Mastering JavaScript from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.