CHAPTER 3

Data

CHAPTER CONTENTS

Introduction

Programs consist of both data and instructions for processing data. Many games, for example, feature players that are moved and drawn. Word processors hold lines of text, which are spellchecked, hyphenated, and positioned to flow around images, tables, footnotes, and other blocks of text. Search engines interpret queries, find matching content, and present results in a useful manner. Photo editors crop, sharpen, rotate, and scale images.

In this chapter we will look at the different kinds of information that programs manipulate. We will see that data may come in primitive forms like numbers and text, as well as in structured forms that look as close to real-life objects as needed. By the end of this chapter, you will be able to express some nontrivial information structures that form the basis for arbitrarily complex scripts.

3.1 Data Types

In JavaScript, there are exactly six *types* of data:

1. *Booleans*, the two values `true` and `false`
2. Numbers, such as 81 and 4.21
3. Text, known in JavaScript as *strings* of characters
4. The special value `undefined`
5. The special value `null`
6. *Objects*

Data types are characterized not only by the values that belong to the type but by the *operations* that are permitted on those values. For example, numbers can be multiplied, subtracted, and exponentiated, while strings can be trimmed, spliced, reversed, and capitalized. You will see all of JavaScript's data types, and a handful of operations, in the remainder of this chapter.

Unlike the previous chapter, which focused was on acclimating you to running scripts without concern for detail, the material we cover here is intentionally quite

technical at times. We have therefore included a fair number of example scripts. Each can be run in your favorite JavaScript environment, and we encourage you to not only run them exactly as presented here, but to experiment by making changes and running the resulting scripts.

3.2 Truth Values

The two values `true` and `false` show up again and again in programming. Here are a few examples you are likely to encounter:

```
var open = true;
var ready = false;
var gameOver = false;
var friendly = true;
var enabled = true;
```

They will also appear as the results of *comparisons*—expressions that compute whether one value is equal to (`===`), not equal to (`!==`), less than (`<`), less than or equal to (`<=`), greater than (`>`), or greater than or equal to (`>=`) another value.

```
alert(137 === 5);   // 137 equal to 5? No, alerts false.
alert(8 !== 3.0);   // 8 not equal to 3.0? Yes, alerts true.
alert(2 <= 2);      // 2 less than or equal to 2? Yes, alerts true.
var x = 16 > 8;     // Truth values can be stored in variables.
alert(x);           // Alerts true.
```

These values are called *Boolean values*.[1] We can apply several operators to these values, namely `&&` (AND-ALSO), `||` (OR-ELSE), and `!` (NOT). If x and y are Boolean values:

- `x && y` is true if and only if x is true **and also** y is true.
- `x || y` is true if and only if x is true **or else** y is true.
- `!x` is true if and only if x is **not** true.

[1] In honor of George Boole, a 19th century mathematician and philosopher.

The reason that && and || are called AND-ALSO and OR-ELSE (as opposed to just AND and OR) will be given in Section 4.3.5.

Let's see the operators in action:

```
alert(4 < 5 && 15 === 6);   // Alerts false.
alert(1 == 2 || 15 > -5);   // Alerts true.
alert(!(3 <= 10));          // Alerts false.
```

You can "compute" with Boolean operators not unlike the way you would compute with numeric operators:

```
var x = 42;
var y = -1;
var bothPositive = x > 0 && y > 0;
var atLeastOneNegative = x < 0 || y < 0;
var exactlyOneNegative = x < 0 !== y < 0;
var atLeastOneNonPositive = !bothPositive;
```

Table 3.1 summarizes the workings of these three operators for all combinations of Boolean operands.

x	y	x && y	x || y	!x
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Table 3.1
Boolean Operators

Review and Practice

1. Evaluate the expression `!(true && !false && true) || false`.
2. Evaluate the expression `false || true && false`.
3. Prove or disprove the following: if `x` and `y` hold Boolean values, then `!(x && y)` is always the same as `(!x || !y)`.
4. Write an expression that is true if and only if the value stored in variable `x` is between 0 and 10 (inclusive).

3.3 Numbers

The next data type we will visit is the number type. You write numbers in JavaScript pretty much as you would expect: `1729`, `3.141592`, and `299792458`. The letter `E` (or `e`) sandwiched between two numbers is read "times 10 to the power":

`3.6288E6`	$\Rightarrow$	3.6288×10^{6}	$\Rightarrow$	3628800
`5.390E-44`	$\Rightarrow$	5.390×10^{-44}		
`4.63e170`	$\Rightarrow$	4.63×10^{170}		

3.3.1 Numeric Operations

Operators on numbers include `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (modulo). The modulo operator computes the remainder after division:

```
alert(48 % 5);          // Alerts 3.
alert(31.5 % 2.125);    // Alerts 1.75.
```

Other operations include `Math.floor(`x`)`, which produces the largest *integer*[2] less than or equal to x; `Math.ceil(`x`)`, which produces the smallest integer greater than or equal to x, `Math.sqrt(`x`)`, which produces $\sqrt{x}$; `Math.pow(`x, y`)`, which

[2]The integers are $\ldots -3, -2, -1, 0, 1, 2, 3 \ldots$.

produces x^y (x to the power y); and `Math.random()`, which produces a random number between 0 (inclusive) and 1 (exclusive).

```
alert(Math.floor(2.8));     // Alerts 2.
alert(Math.floor(-2.8));    // Alerts -3.
alert(Math.ceil(2.8));      // Alerts 3.
alert(Math.ceil(-2.8));     // Alerts -2.
alert(Math.ceil(-5));       // Alerts -5.
alert(Math.sqrt(100));      // Alerts 10.
alert(Math.pow(2.5, 4));    // Alerts 39.0625.
alert(Math.random());       // Alerts something between 0 and 1.
```

The complete set of `Math` operations can be found in Appendix A on page 13.

JavaScript also contains several operators that work on the "internal representation" of integers, namely `~`, `&`, `|`, `^`,`<<`, `>>`, and `>>>`. These are rarely encountered in everyday scripts, so we will not discuss them here.

Review and Practice

1. Evaluate the following expressions: `1 / 8`, `253E-2 * 10`, `36 % 7 + 5`, and `4 - 3 * 10 + 2`.
2. What does `Math.floor(Math.random()*6)+1` produce? (Use a test page or shell, making sure to evaluate this expression several times.)
3. Evaluate the expression `Math.atan(1) + Math.atan(2) + Math.atan(3)`.

3.3.2 Size and Precision Limits

JavaScript numbers, like numbers in most programming languages, are not like the idealized numbers you encounter with every day. For one thing, they need to fit in finitely sized, physical components inside a computing device. Thus, there is a largest finite number (JavaScript's is approximately 1.79×10^{308}),[3] and a smallest finite number (approximately -1.79×10^{308}). Any computation producing a value larger than the largest finite number (or smaller than the smallest) yields the special value `Infinity` (or `-Infinity`).

[3] Or $2^{1024} - 2^{971}$ to be exact.

```
alert(2E200 * 73.987E150);   // Alerts Infinity.
alert(-1e309);               // Alerts -Infinity.
```

Not only are there limits to the *size* of numbers, but their *precision* is limited, too. Computations resulting in numbers that cannot be represented exactly will produce the nearest representable number. This may cause some surprises:

```
alert(12157692622039623539);     // Alerts 12157692622039624000.
alert(12157692622039623539 + 1); // Alerts 12157692622039624000.
alert(1e200 === 1e200 + 1);      // Alerts true.
alert(4.18e-1000);               // Alerts 0.
alert(0.1 + 0.2);                // Alerts 0.30000000000000004.
alert(0.3 === 0.1 + 0.2);        // Alerts false.
```

Many scripts never run into, or can easily tolerate, these approximations. However, sometimes the loss of precision means trouble (think of financial calculations). You should therefore have some sense of where approximations are likely to occur. The following points are particularly relevant:

- The representable numbers are packed most tightly around zero; in fact, more than half of them lie between -1 and 1. The farther you stray from 0, the more spread out they become.

- All integers between -9007199254740992 and 9007199254740992 are represented exactly. (You don't have to memorize those exact values—just remember that "counting" computations are safe within $\pm$ 9 quadrillion, more or less.) Beyond these limits, some, but not all, of the integers will be represented exactly.

- Computations involving (or yielding) very large numbers, very small numbers, or non-integers will often produce non-exact results.

If you ever require the largest representable number, use the expression `Number.MAX_VALUE`. For the smallest, use `-Number.MAX_VALUE`. The expression `Number.MIN_VALUE` gives you the smallest representable number greater than zero, namely 2^{-1074}.

Review and Practice

1. Evaluate the expression `152376357*349982379`. Why is the final digit of the result not a 3?
2. Evaluate the two expressions `(1E200*1E200)/1E200` and `1E200*(1E200/1E200)`, and explain why the answers are not the same.

3.3.3 NaN

The special value `NaN`, meaning *Not a Number*, appears when the result of an arithmetic computation is mathematically undefined:

```
alert(0 / 0);               // Alerts NaN.
alert(Infinity * Infinity); // Alerts Infinity.
alert(Infinity - Infinity); // Alerts NaN.
alert(NaN + 16);            // Alerts NaN.
alert(NaN === NaN);         // Alerts false.
```

The last example above is surprising: `NaN` is not equal to anything, not even `NaN`! To test whether something is not a number, use `isNaN`:

```
alert(0 / 0 === NaN);       // Alerts false.
alert(isNaN(0 / 0));        // Alerts true.
alert(isNaN(2.718281828));  // Alerts false.
alert(isNaN(NaN));          // Alerts true.
alert(isNaN(Infinity));     // Alerts false.
```

We will see more details regarding `NaN` and `isNaN` in Section 3.8.

Review and Practice

1. Explain why `Infinity + Infinity` evaluates to ∞ but `Infinity - Infinity` evaluates to `NaN`. Hint: Consider $(\infty+\infty)-\infty$ and $\infty+(\infty-\infty)$.
2. Evaluate the expression `-2 === Math.sqrt(-2 * -2)`.
3. Evaluate the expression `isNaN(Math.sqrt(-1))`.

3.3.4 Hexadecimal Numerals

Non-negative integers in JavaScript can also be written in *hexadecimal notation*. Hexadecimal numerals count like so: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, . . . , 19, 1A, 1B, . . . , 1F, 20, . . . , 9F, A0, . . . , FF, 100, 101 . . . , FFF, 1000, ... To write an integer in hexadecimal, prefix the numeral with `0x`; for example:

```
alert(0x9);      // Alerts 9.
alert(0x9FA);    // Alerts 2554.
alert(-0xCafe);  // Alerts -51966.
alert(0xbad);    // Alerts 2989.
```

Note that you cannot use fractional parts nor scientific notation with hexadecimal.

Some versions of JavaScript treat integers beginning with a 0 as *octal numerals*. Octal numerals count like so: 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, . . . , 17, 20, . . . , 27, 30, . . . , 77, 100, 101, . . . , 777, 1000, ... In these versions of JavaScript, you would see the following:

```
alert(07);      // Alerts 7.
alert(011);     // Alerts 9.
alert(-02773);  // Alerts -1531.
```

We recommend avoiding octal notation at all costs. We mention it only in case you accidentally type extra zeros at the beginning of a number and find your script producing strange results.

Review and Practice

1. Without using a shell, what is the numeric value of the JavaScript expression `0x10`?
2. Determine if your JavaScript interpreter treats integers with leading zeros as octal.

3.4 Text

A *string* is a sequence of *characters*. In JavaScript, string values are written within double quotes (e.g., `"hello"`) or single quotes (e.g., `'hello'`), and *must fit on one line*.

3.4.1 Characters, Glyphs, and Character Sets

A character is a named symbol, such as:

PLUS SIGN
CYRILLIC SMALL LETTER TSE
BLACK CHESS KNIGHT
DEVANAGARI OM
MUSICAL SYMBOL FERMATA BELOW

Don't confuse a character with a *glyph*, which is a picture of a character. The glyph

K

could represent either the character LATIN CAPITAL LETTER K, GREEK CAPITAL LETTER KAPPA, or CYRILLIC CAPITAL LETTER KA. Similarly, the glyph

Σ

could represent either the character GREEK CAPITAL LETTER SIGMA or SUMMATION SIGN. The glyph

Ø

could represent EMPTY SET, LATIN CAPITAL LETTER O WITH STROKE, or DIAMETER SIGN.

A *character set* is a particular set of characters, each having a unique number called its *codepoint*. JavaScript, like most modern languages, uses the Unicode character set. There is a tradition in Unicode to give the codepoint for each character in hexadecimal. Table 3.2 shows some codepoints for selected Unicode characters.

Codepoint	Character
F1	LATIN SMALL LETTER N WITH TILDE
3B8	GREEK SMALL LETTER THETA
95A	DEVANAGARI LETTER GHHA
F0A	TIBETAN MARK BKA- SHOG YIG MGO
11F4	HANGUL JONGSEONG KAPYEOUNPHIEUPH
13C9	CHEROKEE LETTER QUO
21B7	CLOCKWISE TOP SEMICIRCLE ARROW
265B	BLACK CHESS QUEEN
2678	RECYCLING SYMBOL FOR TYPE-6 PLASTICS
FE7C	ARABIC SHADDA ISOLATED FORM
1D122	MUSICAL SYMBOL F CLEF

Table 3.2

Codepoints for Selected Characters

For the complete codepoint mapping, see `http://www.unicode.org/charts`.

Why do codepoints matter? In JavaScript, you use codepoints to write strings with characters you might not be able to type on your keyboard. For example, the string "Privet" can be given as

```
"\u041f\u0440\u0438\u0432\u0435\u0442"
```

The two characters `\u` followed by four hexadecimal digits stand for the character whose codepoint is given by those digits. You can even use `\x` followed by *two* hexadecimal digits; therefore, you can write the string "Olé" two different ways:

```
"Ol\xc9"
"Ol\u00c9"
```

Some characters are not displayable at all, so the codepoint notation is required. Examples include LEFT-TO-RIGHT MARK (`\u200e`), RIGHT-TO-LEFT MARK (`\u200f`), ZERO WIDTH NO BREAK SPACE (`\ufeff`), and many others. The first two of these characters appear in documents mixing text from languages read left to right (such as English and Spanish) with those read right to left (such as Hebrew and Arabic).

Escape	Description
\'	The single-quote character (used to get the single-quote character in a string surrounded by single quotes)
\"	The double-quote character (used to get the double-quote character in a string surrounded by double quotes)
\x*hh*	Where *hh* is a two-digit hexadecimal value, is the character whose codepoint is that value
\u*hhhh*	Where *hhhh* is a four-digit hexadecimal value, is the character whose codepoint is that value
\n, \t, \b, \f, \r, \v	The characters LINE FEED, CHARACTER TABULATION, BACKSPACE, FORM FEED, CARRIAGE RETURN, and LINE TABULATION
\\	The backslash character itself

Table 3.3

JavaScript Escape Sequences

The backslash character "\" is used for more than just specifying characters by their codepoint; it combines with subsequent characters as shown in Table 3.3 to form what are called *escape sequences*.

The linefeed character, \n, causes the rest of the string to be written on the next line, and the tab character, \t, helps to align text in columns. Here's a fun one-line script that alerts a string with tabs and newlines; Figure 3.1 shows the output.

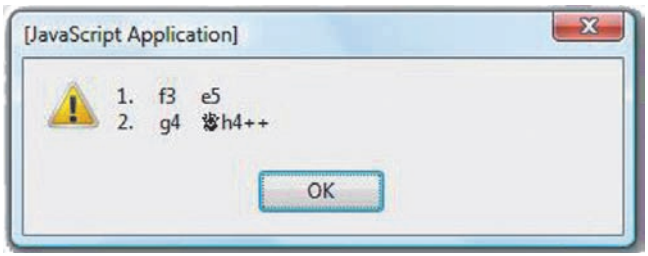

FIGURE 3.1

A string with tabs and newlines.

```
alert("1.\tf3\te5\n2.\tg4\t\u265bh4++");
```

The \u notation only works in JavaScript strings; to get characters in HTML documents, you use a different notation: surround the (hexadecimal) codepoint with &#x and a semicolon (";"). Examples:

Character	JavaScript	HTML
BLACK CHESS QUEEN	\u265b	♛
DIE FACE-1	\u2680	⚀
DIE FACE-2	\u2681	⚁
DIE FACE-6	\u2685	⚅
TIBETAN LETTER DZHA	\u0f5c	ཛྷ

Let's create a slightly fancier die-rolling program to display die face characters directly in the HTML page. We will define a Roll button that when clicked generates a random integer in $\{0, 1, 2, 3, 4, 5\}$. Since the die face characters in Unicode have codepoints 2680 through 2685, the HTML is fairly easy to form. Here's a quick implementation:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Die Rolling</title>
    <style>div#die {font-size: 800%}</style>
  </head>
  <body>
    <div><input id="roller" type="button" value="Roll" /></div>
    <div id="die"></div>
    <script>
      document.getElementById("roller").onclick = function () {
          document.getElementById("die").innerHTML =
              "&#x268" + Math.floor(Math.random() * 6) + ";";
      }
    </script>
  </body>
</html>
```

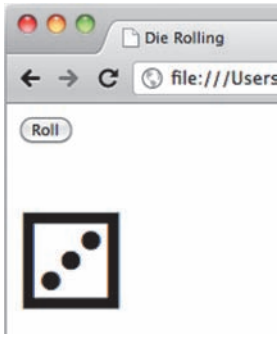

FIGURE 3.2
A screenshot of the die-rolling script.

This script introduces the HTML `style` element, which we will cover in some detail in Chapter 6. For this document, we have introduced one style rule, which says, "For any `div` element whose `id` attribute is `die`, render text eight times larger than usual." Figure 3.2 shows a screenshot after rolling a 3. Your results may vary, however. Just because certain characters are defined in Unicode does not mean every browser is capable of displaying glyphs for them. Whether or not you (or your users!) can see particular characters depends on the *fonts* that are installed on the system.

Please note that our program was very small, so we can get away with embedding the script inside the HTML document. Remember to define scripts in separate files when your applications are longer (as discussed in Section 2.2.4). The same can be said for style rules. Because our page was very small, we placed style rules in the document head. In larger pages, style rules would appear in a separate file so that the document structure (HTML), its presentation (styles), and its behavior (JavaScript) are each in their own place. This *separation of concerns* is one of the many important concepts in software engineering.

Review and Practice

1. Look up, in a Unicode book or online reference, the codepoints for the following characters: COMMA, ORIYA DIGIT SEVEN, CANADIAN SYLLABICS LWII, APPROACHES THE LIMIT, BRAILLE PATTERN DOTS-456, COFFIN, TETRAGRAM FOR RITUAL, and CJK STROKE HZG.

2. Write a one-line script to alert (the song title) "`Takogo kak Putin`."

3. Enter and run the die-rolling script from this chapter. Can your browser display the die face characters? If not, what is shown in their place?

3.4.2 String Operations

JavaScript provides dozens of string operations, such as finding the length (number of characters) in a string,[4] translations to upper- and lowercase, and replacing parts of the string.

```
alert("Hello, there".length);              // Alerts 12.
alert("Hello, there".toLowerCase());       // "hello, there".
alert("Hello, there".toUpperCase());       // "HELLO, THERE".
alert("Hello, there".replace("ello", "i")); // "Hi, there".
```

Sometimes you would like to know where in a string a certain character can be found, or find out which character is at a given position. In JavaScript, the first character in a string is at *index* 0, the second at 1, the third at 2, and so on. The character at position p within string s is found with the expression s`.charAt(`p`)`. Text within a string can be located with `indexOf` and `lastIndexOf`. The expression s`.substring(`x, y`)` produces a string consisting of all characters of string s from index x up to but **not including** the character at index y.

```
"Some text".charAt(7)          ⇒  "x"
"Some text".indexOf("me")      ⇒  2
"Some text".lastIndexOf("e")   ⇒  6
"Some text".substring(3, 7)    ⇒  "e te"
```

The + operator *concatenates* two strings:

```
"dog" + "house"  ⇒  "doghouse"
"2" + "2"        ⇒  "22"
```

Here is an example using substring and concatenation to rewrite a phone number from one format to another:

```
var phone = "(800) 555-1212";
var area = phone.substring(1, 4);
var prefix = phone.substring(6, 9);
var suffix = phone.substring(10, 14);
alert(area + "." + prefix + "." + suffix); // 800.555.1212
```

[4] We should be honest and say "more or less" here, because characters with codepoints greater than FFFF are counted twice. See Appendix C for details.

One extremely important aspect of JavaScript strings is that they are *immutable*. This means that you cannot change the characters within a string; nor can you change the string's length. For example, what do you think this code does?

```
var s = "Hello";
s.toUpperCase();
alert(s);
```

This alerts `Hello` because the second line simply performs a computation to uppercase a string but does nothing with the result! If the intent is to alert the uppercase version of the string, you can assign the uppercase string to a new variable, or alert the `s.toUpperCase()` expression directly. Try it!

Review and Practice

1. What is `"abcdef".substring(3,4)`?
2. Write a one-line script to alert the text "The backslash character (\) is cool."
3. Evaluate the expression `"dog".charAt(2).charAt(0)` and explain the result.

3.5 Undefined and Null

Usually in programming we are interested in capturing actual knowledge, such as how much something costs (a number), whether or not a player in a game is active (a Boolean), or the name of your supervisor (a string). But sometimes we want to capture the nonexistence of data, or the uncertainty of data. Let's look at the supervisor example more deeply. What might you say about your supervisor?

1. I have a supervisor and her name is Alice.
2. I definitely have no supervisor.
3. I may or may not have a supervisor; I really don't know.
4. I do know whether I have a supervisor, but I don't care to make this information public.

JavaScript gives you the special value `null` as a way to express the second scenario, and `undefined` for the third and fourth.

```
var supervisor = "Alice";   // The supervisor is Alice.
var chief = null;           // Absolutely, positively, NO chief.
var assistant = undefined;  // There might be an assistant.
```

Review and Practice

1. Explain in your own words the difference between `undefined` and `null`.
2. Why might it be a bad idea to use strings such as `"NONE"` and `"UNKNOWN"` instead of `null` and `undefined`, respectively?

3.6 Objects

3.6.1 Object Basics

In JavaScript, any value that is neither a Boolean, a number, a string, `null`, nor `undefined` is an *object*. Objects have *properties*, and properties have *values*. Property names can be strings (which you might have to place in quotes) or non-negative integers (0, 1, 2, ...). Property values may be objects as well, enabling complex structures to be defined. An *object literal* is an expression defining a new object; several examples follow:

```
var dress = {
    size: 4,
    color: "green",
    brand: "DKNY",
    price: 834.95
};
```

```
var location = {
    latitude: 31.131013,
    longitude: 29.976977
};
```

```
var part = {
    "serial number": "367DRT2219873X-785-11P",
    description: "air intake manifold",
    "unit cost": 29.95
};
```

```
var p = {
    name: { first: "Sen", last: "O'Brien" },
    country: "Ireland",
    birth: { year: 1981, month: 2, day: 17 },
    kidNames: { 1: "Ciara", 2: "Bearach", 3: "Mirad", 4: "Aisling" }
};
```

After defining an object, you may access its properties with either a dot or square brackets.

```
p.country           ⇒  "Ireland"
p["country"]        ⇒  "Ireland"
p.birth.year        ⇒  1952
p.birth["year"]     ⇒  1952
p["birth"].year     ⇒  1952
p["birth"]["year"]  ⇒  1952
p.kidNames[4]       ⇒  "Aisling"
p["kidNames"][4]    ⇒  "Aisling"
```

The dot-notation for properties is a bit more concise but cannot be used with integer properties (i.e., you can't say "`a.1`"), nor can it, in ES3,[5] be used if the property is a JavaScript reserved word (see page 2). In these cases, the bracket notation is required (e.g., `a[10]`, `a["var"]`). Bracket notation is also required for property names containing spaces or certain other non-alphanumeric characters (e.g., `part["serial number"]`).

[5]Recall from Section 2.5 that ES3 is the old specification for JavaScript.

The properties of an object are not fixed; in fact, you can add and even delete properties, as in the following example:

```
var dog = {};               // An object with no properties.
dog.name = "Krl";           // Now the object has one property.
dog.breed = "Rottweiler";   // Now the object has two properties.
delete dog.name;            // Now one property again.
```

When JavaScript is used for web applications, the elements that make up the web page are objects with their own properties. The following little web program demonstrates this; it makes a happy face jump around the browser by randomly placing it at a new random position every 2 seconds.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Jumping Happy Face</title>
    <style>
      div#face {font-size: 500%; position: absolute;}
    </style>
  </head>
  <body>
    <div id="face">&#x263a;</div>
    <script>
      var style = document.getElementById("face").style;
      var move = function () {
        style.left = Math.floor(Math.random() * 500) + 'px';
        style.top = Math.floor(Math.random() * 400) + 'px';
      }
      setInterval(move, 2000);
    </script>
  </body>
</html>
```

The happy face itself is simply the character U+263A embedded in an HTML `div` with the id "face." This element has a `style` property that is itself an object with properties that include `position`, `left`, and `top`. The script arranges, via

`setInterval`, to run the `move` function[6] every 2000 milliseconds, or 2 seconds. The values of the style properties `left` and `right` are strings that can take various forms, one of which represents pixel offsets in the browser window and is denoted with values such as `288px`. Our program assigns new style values each time the `move` function is run. JavaScript detects the style changes and refreshes the browser window. The program will run until you close the browser window or load another page.

Review and Practice

1. When must an object's properties be accessed with the square bracket notation as opposed to dot-notation?
2. What does the script

```
var pet = {name: "Oreo", type: "Rat"};
alert(pet[name]);
```

   output? Why?

3.6.2 Understanding Object References

Objects differ from the other five types of values, collectively called *primitive values*, in two very important ways. Although quite technical, these two differences are so essential to understanding how to use objects correctly and efficiently that they must be memorized. The first is:

The *value* of an object expression is not the object itself, but a *reference* to it.

This can only be explained with pictures. Figure 3.3 shows how primitive values are stored directly inside variables (see variable `a` in the figure), while objects are not. Variables contain references to objects (see variable `b`).

Because object values are actually references, the assignment of an existing object value to a variable makes *a copy of the reference*, not of the object itself;

[6] Although functions will not be officially covered until Chapter 5, you should, we hope, be able to follow the general idea behind the code.

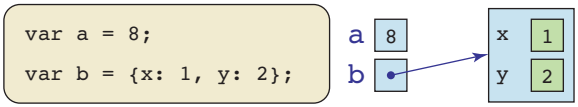

FIGURE 3.3

Object values are stored as references.

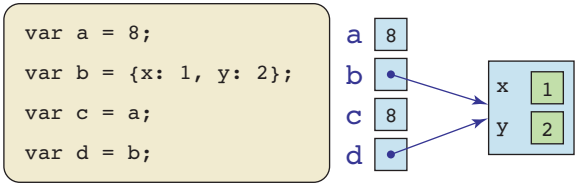

FIGURE 3.4

Assignment.

that is, *no new object is created*. If you think about it, assignment doesn't really work any differently for primitives than it does for objects; assignment between variables is always copying what is in one box into another box, regardless of whether that box holds a number or an arrow. Figure 3.4 illustrates the assignment of both primitives and object references, and should be studied carefully.

The second important way in which objects differ from the other types of values is:

Every evaluation of an object literal creates a brand new object.

This fact is illustrated in Figure 3.5, where a script declares three variables and creates two objects. Although the two objects have the exact same properties with the same values, there were two distinct object literals in the script, and therefore two distinct objects were created.

The fact that variables hold references and not objects not only requires us to think carefully about assignment, but also equality testing. To evaluate the expression `x===y`, we ask "do `x` and `y` have the same value?" For variables referring to objects, we must ask whether each reference points to the *same object*. In Figure 3.5 we have `a===b` but `a!==c`. In the latter case we have two objects that look alike, but they are still two distinct objects.

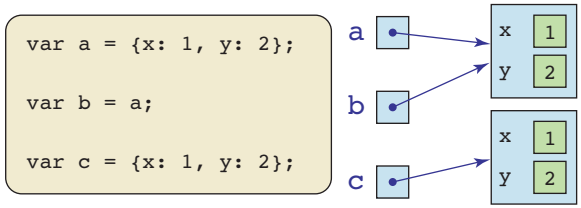

FIGURE 3.5

Three variables and two objects.

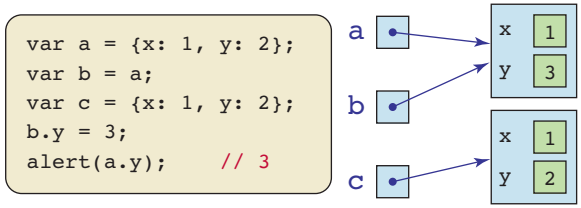

FIGURE 3.6

Updating a property of a shared object.

Because an object can be simultaneously referred to by more than one variable, either variable can be used to update the object's properties, and either can be used to see these updates. See Figure 3.6.

Review and Practice

1. Draw pictures that illustrate the variables `dress`, `location`, `part`, and `p` and their values given at the beginning of Section 3.6.1.
2. How does assignment of primitive values differ from that of reference values, if at all?
3. Evaluate the expression `{x:1,y:2} === {x:1,y:2}` and explain the result.

3.6.3 Object Prototypes

In Section 3.6.1 we created one dress, one location, one part, and one person object. What if you needed several of the same kind of object? You can define them straight up, of course, as in this collection of circles:

```
var c1 = {x: 4, y: 0, radius: 1, color: "green"};
var c2 = {x: 4, y: 0, radius: 15, color: "black"};
var c3 = {x: 0, y: 0, radius: 1, color: "black"};
```

But JavaScript lets you do something more elegant: you can create a prototypical circle from which other circles are based. Every JavaScript object has a hidden link to another object, called its *prototype*. When trying to read a nonexistent property in an object, JavaScript looks in the object's prototype. If the property is not in the prototype, it checks the prototype's prototype, and so on. The last object on this *prototype chain* should have the value `null` for its hidden link. If the property is not found anywhere on the chain, you will get an error.[7] Figure 3.7 shows a prototype circle, which is centered at $(0, 0)$, with radius 1, and colored black; we have placed another circle at $(4, 0)$ and colored it green. The hidden prototype link gives the new circle (`c1`) the same y-value and radius as the prototype. In fact, circle `c1` has four properties: `x` and `color` are called *own properties*, while `y` and `radius` are *inherited properties*.

You create an object from a prototype with `Object.create`. This is an ES5 operation, so old (ES3) JavaScript engines have to use another technique, which we cannot reveal until Chapter 5. Here's the ES5 code that creates the two objects in Figure 3.7:

```
var protoCircle = {x: 0, y: 0, radius: 1, color: "black"};
var c1 = Object.create(protoCircle);
c1.x = 4;
c1.color = "green";
// Note that c1.y === 0 and c1.radius === 1 (inherited properties)
```

Prototypes really shine when you derive many objects from them; in Figure 3.8 we have made two more circles. One of the two new circles has no properties of its own (pun intended); it happily inherits *all* the properties from the prototype.

[7] Technically, a `ReferenceError` will be thrown; these will be discussed in Chapter 4.

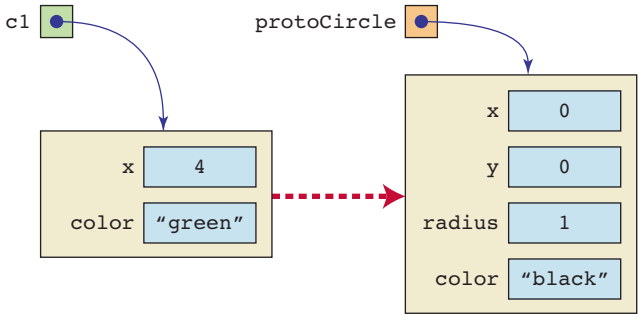

FIGURE 3.7

An object and its prototype.

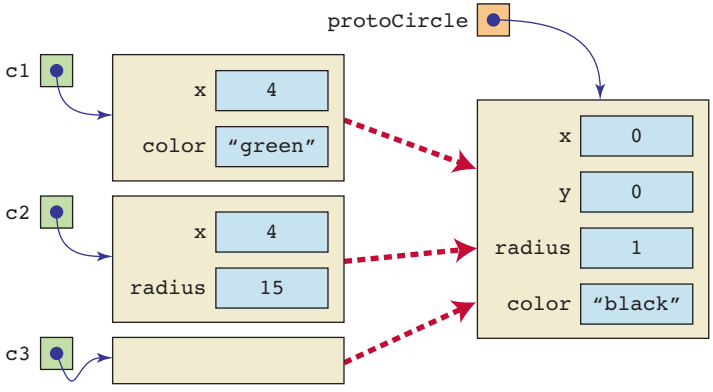

FIGURE 3.8

Multiple objects sharing a prototype.

Review and Practice

1. Define the terms *own property* and *inherited property* in your own words.
2. Write the code to create circles `c2` and `c3` in Figure 3.8.
3. In Figure 3.8, after running `protoCircle.radius = 5`, what would happen to the value of `c1.radius`? To `c2.radius`?

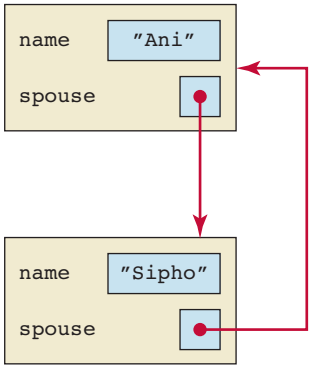

FIGURE 3.9
Objects that reference each other.

3.6.4 Self-Referential Objects

It is possible for an object to have a property whose value is a reference to itself, or for two objects to refer to each other, as in Figure 3.9. Notice that in such cases, it is not possible to describe all of the objects purely with object literals. (Try it!)

In such cases we can create partial objects with object literals and fill in the rest with assignment.

```
var mom = {name: "Ani"};
var dad = {name: "Sipho", spouse: mom};
mom.spouse = dad;
```

Review and Practice

1. Draw a picture of the objects resulting from the following declaration:

   ```
   var p1 = {name: "Alice"};
   var p2 = {name: "Bob", manager: p1};
   p1.manager = p1;
   ```

2. What is the value of `p2.manager.manager.manager.name` in the previous question?

3.7 Arrays

An *array* is a special kind of object whose properties are consecutive non-negative integers from 0 to some limit, together with a property called `length`. Arrays are special because you do not create them with regular object literals, but instead with a special syntax:

```
var a = [];
var b = [8, false, [[null, 9]];
var days = ["p\u014d\u02bbakahi", "p\u014d\u02bbalua",
    "p\u014d\u02bbakolu", "p\u014d\u02bbah\u0101",
    "p\u014d\u02bbalima", "p\u014d\u02bbaono",
    "l\u0101pule"];
```

These arrays are illustrated in Figure 3.10.

This special syntax actually creates the properties 0, 1, 2, ... , and `length`. The length property is special, too: assigning to it may grow or shrink the array; newly added properties, if any, are set to `undefined`. An array may also be grown by assigning to a position beyond the end of the array.

```
var a = [9, 3, 2, 1, 3]; // a[0] is 9, a.length is 5, etc.
a[20] = 6;               // a[5] through a[19] now undefined.
alert(a.length);         // Alerts 21.
a.length = 50;           // a[21] through a[49] all undefined.
a.length = 3;            // a is now [9, 3, 2].
```

You can also create arrays by `split`-ting up a string across a given separator, by extracting a `slice` from an existing array, or `concat`-enating two arrays. You can also `join` array elements together into a string while introducing a separator:

```
var s = "A red boat";
var a = s.split(" ");        // a is ["A", "red", "boat"].
var b = [9, 3, 2, 1, 3, 7];
var c = b.slice(2, 5);       // c is [2, 1, 3].
var d = c.concat(a);         // d is [2, 1, 3, "A", "red", "boat"].
alert(d.join("**"));         // Alerts 2**1**3**A**red**boat.
```

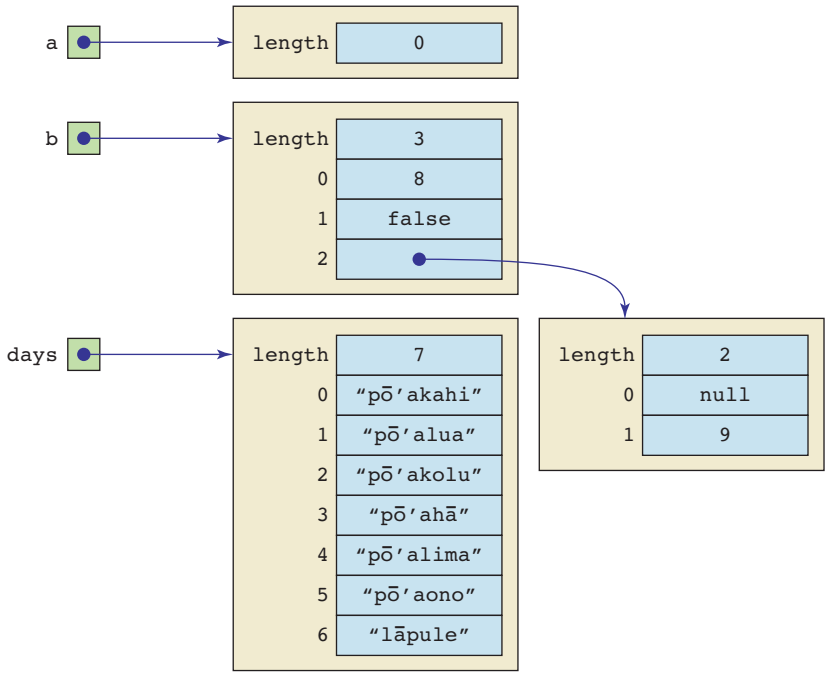

FIGURE 3.10

Arrays.

Note that the `slice` operation, when asked to create an array from the slice of `b` from positions 2 to 5, returned the array `[b[2], b[3], b[4]]`. This operation always includes the element at the starting position and excludes the one at the ending position; this is exactly the same behavior we saw with the `substring` operation back on page 69. That is, `a.slice(x,y)` creates a new array with elements `a[x]` through `a[y-1]`, by design. There are two other forms of the slice operation:

```
var a = [9, 4, 1, 7, 8];
var b = a.slice(2);       // from index 2 to the end ([1, 7, 8)
var c = a.slice();        // all indices - makes a copy of a
```

The `split` operation does not change the string being split; nor do the `slice`, `concat`, and `join` operations change the arrays they operate upon. However, several operations *do* change the array objects themselves. You can add items to the end with `push` and to the beginning with `unshift`. Remove from the end with `pop`

and from the front with `shift`. You can even `reverse` and `sort` arrays. Operations that modify the objects on which they act are called *mutators*.

```
var a = [];             // a is an array of length 0.
var b = [3, 5];         // b has length 2.
b.push(2);              // Now b is [3, 5, 2].
b.unshift(7);           // Now b is [7, 3, 5, 2].
a.push(3, 10, 5);       // Now a is [3, 10, 5].
a.reverse();            // Now a is [5, 10, 3].
alert(a.pop());         // Alerts 3 and changes a to [5, 10].
alert(a.shift());       // Alerts 5 and changes a to [10].
b.push(a[0], 1);        // b is now [7, 3, 5, 2, 10, 1].
b.sort();               // b is now [1, 10, 2, 3, 5, 7].
```

What? How did 10 get between 1 and 2? JavaScript put it there because the `sort` operation treats all array elements as strings, even if the array contains numbers, Booleans, objects, or anything else. Sorting strings means putting them in alphabetical order, and the string `"1"` is less than `"10"`, which is less than `"2"` (just like `"foot"` is less than `"football"`, which is less than `"goal"`). You *can* sort arrays using a numeric ordering as well; we will see how later in the text.

You can create arbitrarily complex structured data by placing arrays within object literals and creating arrays of object literals; for example:

```
var song = {
    title: "In My Head",
    track_number: 10,
    album: "Rock Steady",
    artist: "No Doubt",
    authors: ["Gwen Stafani", "Tony Kanal", "Tom Dumont"],
    duration: 205
};

var triangle = [{x: 0, y: 0}, {x: 3, y: -6}, {x: -4, y: -1.5}];
```

In practice, you will use regular objects to describe (individual) things, such as dresses, people, songs, and points, and you will use arrays to describe *lists* of

things, such as songs on an album or the points of a polygon. Arrays are not the only way to form a collection of values; several exercises at the end of this chapter will introduce you to ways of wiring together objects to make different kinds of collections.

Review and Practice

1. Draw a picture of the variables `song` and `triangle` defined at the end of this section, and the objects they reference.
2. Draw a picture of the array referred to by the variable `a` after executing the following: `var a = [1,2,3,4]; a.unshift(a.pop());`
3. Draw a picture of the array referred to by the variable `a` after executing the following: `var a = [1,2,3,4]; a.push(a.shift());`
4. Describe the split and join operations in your own words.

3.8 Type Conversion

3.8.1 Weak Typing

Now that we have seen each of JavaScript's six types, let's take a closer look at the operations on those types. We have seen a few operators so far, including:

Boolean `&&` *Boolean*	*number* `/` *number*
Boolean `||` *Boolean*	*number* `%` *number*
`!` *Boolean*	`Math.sqrt(` *number* `)`
`-` *number*	*string* `+` *string*
number `+` *number*	*string*`.toUpperCase()`
number `-` *number*	*string*`.indexOf(` *number* `)`
number `*` *number*	*object* `[` *string* `]`

What happens if we give an operator one or more values of the "wrong" type? Let's experiment. Suppose a number is expected:

```
7 * false      ⇒ 0
7 * true       ⇒ 7
7 * "5"        ⇒ 35
7 * " 5 "      ⇒ 35
7 * " "        ⇒ 0
7 * "dog"      ⇒ NaN
7 * null       ⇒ 0
7 * undefined  ⇒ NaN
7 * {x: 1}     ⇒ NaN
```

Suppose a Boolean is expected:

```
! 5          ⇒ false
! 0          ⇒ true
! "dog"      ⇒ false
! ""         ⇒ true
! " "        ⇒ false
! null       ⇒ true
! undefined  ⇒ true
! {x: 1}     ⇒ false
```

And finally, suppose a string is expected:

```
"xyz" + false      ⇒ "xyzfalse"
"xyz" + true       ⇒ "xyztrue"
"xyz" + 7          ⇒ "xyz7"
"xyz" + null       ⇒ "xyznull"
"xyz" + undefined  ⇒ "xyzundefined"
"xyz" + {x: 1}     ⇒ xyz[object Object]
"xyz" + [1, 2, 3]  ⇒ xyz1,2,3
```

Here we see that JavaScript generally does not complain when operators are given values of the wrong type; instead, it tries to accommodate the programmer

by performing *type conversions* to treat the wrongly typed value in a way that makes sense. It turns out that:

- When a number is expected: `false` is treated as 0, `true` as 1, strings as the number they look like, `null` as 0, and `undefined` as `NaN`. For strings, leading and trailing spaces are ignored, and a string that is empty or composed entirely of whitespace is treated as 0. A string that does not look like a number at all is treated as `NaN`. For an object x, JavaScript evaluates x`.valueOf()`.

- When a Boolean is expected: `0`, the empty string (`""`), `null`, `undefined`, and `NaN` (as well as `false` itself) are treated as false, while all other values are treated as true. The former values are called *falsy*, and the latter *truthy*.

- When a string is expected: JavaScript usually finds something reasonable, as you can see in the preceding examples. For an object x, JavaScript evaluates x`.toString()`.

We will provide a more detailed explanation of `valueOf` and `toString` on page 24. There is also a little detail we have to address about `&&` and `||`: they don't exactly expect Booleans. We will see why in Section 4.3.5.

Because of its penchant for implicit type conversions, JavaScript is called a *weakly typed* programming language. In a strongly typed programming language, expressions formed with values of the wrong type cause errors: scripts containing such ill-formed expressions will either not even be allowed to run, stop working altogether, or throw an exception[8] when the offending expression is evaluated.

Sometimes, these automatic conversions result in some surprises. One bit of odd behavior concerns `isNaN`, which you will recall is used to tell if something is not a number. You might expect Booleans, strings, and `null`, to not be numbers, but experimentation shows us that `isNaN` doesn't work that way:

```
alert(isNaN(true));     // false, because true converts to 1
alert(isNaN(null));     // false, because null converts to 0
alert(isNaN("water"));  // true, and sensibly so
alert(isNaN("100"));    // false, because "100" converts to 100
```

[8] Exceptions will be covered in Section 4.5.2.

This means you should read `isNaN` as "cannot be converted into a number." A second surprise is the following confusion between numbers and strings that occurs very often when one is learning JavaScript:

```
var x = prompt("Enter a number");
var y = prompt("Enter another number");
alert(x + y);           // Concatenation, not arithmetic addition!
```

Entering 2 for each prompt will alert 22, because the result of evaluating a `prompt` is a string, and the `+` operator is (unfortunately) defined on strings as well as numbers.[9] Alerting `x - y` would have done numeric subtraction, however. The operator `-` is defined only on numbers, so JavaScript would have converted both string inputs. Multiplication and division are also "safe"; regardless, you must be always on your guard when it comes to numbers and strings.

3.8.2 Explicit Conversion

Because of the potential for string–number confusion, many JavaScript programmers prefer to make the conversion of strings to numbers more *explicit* in the code. This can be done in several ways:

```
"3.14" - 0          ⇒  3.14
"3.14" * 1          ⇒  3.14
"3.14" / 1          ⇒  3.14
+"3.14"             ⇒  3.14   [Fastest]
Number("3.14")      ⇒  3.14   [Clear, but slow]
parseFloat("3.14")  ⇒  3.14
```

The first three expressions work because `-`, `*`, and `/`, being numeric operators will convert the string to a number before applying the (admittedly useless) operation. The fourth expression also uses a numeric operator, called *unary plus*. When

[9] This is considered by many to be a major design flaw in JavaScript. In fact, many other programming languages are very careful not to use `+` for strings; for example, PHP and Perl use the dot (`.`), ML uses the carat (`^`), and SQL uses the double pipe (`||`).

applied to a number, this operator does not really do anything, unlike its friend *unary minus*:

```
+4  ⇒  4    [Unary Plus]
-4  ⇒  -4   [Unary Minus]
```

However, the fact that unary plus expects a number means that if given a string, JavaScript will convert that string to a number. The use of `+` for converting strings to numbers is rather cryptic, but the technique is convenient and not uncommon. Such handy programming forms are called *idioms*: they are not obvious to "outsiders" and must be learned (just like idioms in human languages).

Our earlier numeric addition script now becomes:

```
var x = +prompt("Enter a number");
var y = +prompt("Enter another number");
alert(x + y);           // Arithmetic addition (2+2=4)
```

You are encouraged to try this script and compare it with the previous attempt. What happens when you enter non-numeric inputs?

What about the forms `Number(`*s*`)` and `parseFloat(`*s*`)` where *s* is a string? The former looks *clearer* (more readable) than the idiomatic form:

```
var x = Number(prompt("Enter a number"));
var y = Number(prompt("Enter another number"));
alert(x + y);           // Arithmetic addition (2+2=4)
```

However, many programmers shun `Number` because its use is *inefficient*: the JavaScript engine does too much work in running the code, slowing the script down and consuming more memory than necessary. It turns out that `Number` does not actually make a primitive number, but rather an object with a number inside of it, which at least (thanks to weak typing) can be pulled out when the object is used when a number is expected. The JavaScript engine does more work for objects than for primitives; it has to find space for objects when they are created, and it needs to eventually get rid of them when they are no longer needed. There are times, however, when readability is more important than efficiency; we will discuss these issues further in Section 7.6.1.

We can also explicitly convert strings to numbers with `parseFloat` and its cousin `parseInt`. These operations produce numbers from the beginning of a

string, not necessarily the whole string, though leading and trailing spaces are ignored:

```
alert(parseFloat("23.9"));        // Alerts 23.9.
alert(parseFloat("5.663E2"));     // Alerts 566.3.
alert(parseFloat("    8.11  "));  // Alerts 8.11.
alert(parseFloat("52.3xyz"));     // Alerts 52.3.
alert(parseFloat("xyz52.3"));     // Alerts NaN.
alert(parseFloat("3 .5 .6"));     // Alerts 3.
```

The `parseInt` operation produces numbers without fractional parts; in fact, the `Int` in `parseInt` means, of course, "integer." Numbers with fractional parts allowed are known in computing circles as *floats*.[10]

```
alert(parseInt("23.9"));        // Alerts 23.
alert(parseInt("5.663E2"));     // Alerts 5.
alert(parseInt("5.663E7"));     // Alerts 5.
alert(parseInt("    8.11  "));  // Alerts 8.
alert(parseInt("52.3xyz"));     // Alerts 52.
alert(parseInt("xyz52.3"));     // Alerts NaN.
```

We can use `parseInt` to deal with numerals in any base between 2 and 36, which we will simply illustrate by example and then move on (for details on numeric bases, see Appendix B):

```
alert(parseInt("75EF2", 16));   // Alerts 483058.
alert(parseInt("50", 8));       // Alerts 40.
alert(parseInt("110101", 2));   // Alerts 53.
alert(parseInt("hello", 30));   // Alerts 14167554.
alert(parseInt("36", 2));       // Alerts NaN.
```

We will rarely, if ever, see `parseInt` and `parseFloat` in the remainder of this text.

[10]Why exactly are they called floats? The idea is that because the decimal point can appear anywhere within the number, it's as if it is allowed to "float around."

Review and Practice

1. Evaluate the following: `"5"+5`, `5+"5"`, `"5"*5`, `5*"5"`, `"5"*"5"`, `3+null`, `"3"+null`, `"dog" + "house"`, and `"dog" - "house"`.

2. Based on your answer to the previous question, give the general rule regarding how JavaScript determines whether to treat `+` as numeric addition or string concatenation.

3.8.3 Loose Equality Operators

Since JavaScript, being weakly typed, tries to make values "match up" with each other typewise before applying operations such as `+`, `-`, `*`, `/`, `<`, and so on, it stands to reason that it might play the same type conversion game with equality operators. That is, if you would like to know whether x equals y, JavaScript's philosophy would be to try to find type conversions to make these values comparable. As it turns out, JavaScript gives you two ways to test equality: one with this implicit type conversion, and one without.

The `===` operator compares two expressions and produces `true` if and only if the two expressions have the same value and have the same type,[11] while `!==`, as you know, produces the opposite truth value from `===`. These are known as the *strict equality operators*. JavaScript's other equality operators, `==` and `!=`, have been described as the "evil cousins" [Cro08a, p. 109] of the strict operators. The `==` operator does the type conversions before testing but goes overboard with its conversions, and should probably be avoided.

The rules for `==` are well defined, even if hard to remember. They are given in full in the official JavaScript specification [ECM09, p. 80], but we can summarize the rules here. To evaluate x `==` y where x and y do not have the same types, JavaScript performs type conversions to attempt to make them comparable. In particular:

1. If one of x and y is a string and the other is a number, JavaScript will work with the numeric conversion of the string.

2. If one is a Boolean and the other is not, JavaScript will work with the numeric conversion of the Boolean.

[11] The one exception is that `NaN` is not equal to anything, not even `NaN`.

3. If one is an object and the other is a string or number, JavaScript will work with the conversion of the object into a string or number (we will see how this is done later in the text).

4. Finally, for some unknown reason, `undefined == null` and `null == undefined`.

Because of the complexity of these rules, we prefer the strict equality operators `===` and `!==`. Although the loose operators provide some "convenient coding shortcuts," such as automatically performing string to number conversions that we spent so much time discussing in the previous section, the fact remains that their behavior is quirky and hard to memorize.

Use === and !== instead of == and !=

Review and Practice

1. Evaluate the following (using a shell), and provide an explanation for your results: `"7" == 7`, `"7" === 7`, `0 == " "`, `0 === " "` `"0" == " "`, `"0" === " "`, `null == undefined`, `null === undefined`, `null == false`, `null === false`, `1 == true`, `4 == true`, `" " == false`.

3.9 The `typeof` Operator*

Occasionally you may find yourself with a value whose type you need to determine (it can happen). JavaScript's quirky `typeof` operator returns a string that describes the type of an expression. We say it is quirky because the value it returns sometimes makes sense and sometimes does not.

```
typeof 101.3       ⇒  "number"
typeof false       ⇒  "boolean"
typeof "dog"       ⇒  "string"
typeof {x:1, y:2}  ⇒  "object"
typeof undefined   ⇒  "undefined"
typeof null        ⇒  "object"
typeof [1, 2, 3]   ⇒  "object"
typeof alert       ⇒  "function"
```

There is no excuse for the type of `null` being `"object"`. It *does* make sense for arrays to have the type `"object"`, since after all, arrays are objects, but one wonders why functions, another kind of object (which we will see in Chapter 5), are treated specially by the `typeof` operator. It's just one of those things.

Review and Practice

1. Explain the output of the following:

```
var x = 2;
alert(typeof x + typeof "x");
```

2. Evaluate `typeof Infinity` and `typeof NaN` in a test page or shell.

Chapter Summary

- An expression is a fragment of code that is evaluated to produce a value.
- Every JavaScript value is either `undefined`, `null`, a number, a Boolean, a string, or an object.
- JavaScript numbers have size and precision limits, so many computations return approximate results.
- The expression `x === y` computes whether `x` and `y` are equal, while `x = y` assigns the value of `y` to `x`. The expression `x == y` is another kind of equality test but should generally be avoided because its behavior, while completely specified in the official language definition, is often unexpected.
- Objects in JavaScript contain a set of zero or more properties. An object's value is really a reference to the object, and it is not uncommon for two or more variables to contain references to the same object.
- An array is a special kind of object containing a collection of values, indexed from 0. Arrays also have a special `length` property. Updating the array may change the `length` property automatically, and updating the `length` property may cause the array to grow or shrink.

- JavaScript is a weakly typed language, meaning that most of the time that operators are given values of the wrong type, these values will be converted into values of the right type to allow the computation to proceed.
- The values 0, `NaN`, `null`, `undefined`, and the empty string (`""`) will be interpreted as false if used in a context where a Boolean value is expected. These values, together with `false`, are called *falsy*; all other values are *truthy*.
- There are many ways to make a string to number conversion explicit in JavaScript, including the unary plus idiom and the `parseFloat` and `parseInt` operations.
- JavaScript has 52 operators, though many of them are very rarely used.

Exercises

1. Read an article on George Boole. Then read an article on Claude Shannon. Discuss how the world might be different without their ideas.
2. Let $x = 10$, $y = 4$, $b =$ false. Evaluate the following expressions:
   (a) `x * y > 25 && !b || y % -3 !== 22`
   (b) `y * 4 === 2 || (b !== true)`
3. Show that when a and b are variables holding Boolean values, `!a && !b || a && b` always evaluates to the same result as `a === b`.
4. We saw in this chapter that `Math.sqrt(100)` evaluates to 10. The use of the dot makes it appear that `Math` is an object and `sqrt` is one of its properties. Is this true? Evaluate `Math["sqrt"](100)` in your favorite JavaScript environment and see if the result supports or refutes this hypothesis.
5. Use a shell to evaluate `~22`, `~105`, `~(-28)` and a few other expressions involving `~` applied to whole numbers. Can you come up with a general rule for what this operator seems to do?

6. Execute the following script:

```
var celsius = prompt("Enter a temperature in \u00b0C");
var fahrenheit = 1.8 * celsius + 32;
alert(celsius + "\u00b0C = " + fahrenheit +  "\u00b0F");
```

   giving inputs of 0, 37, −40, and 100. Then execute with inputs such as `dog`, `2e600`, `3ffc`, and `Infinity`. For each execution, state whether your observed result made sense; if not, state why the result differed from your expectation.

7. Write a script similar to one in the previous exercise that converts a user-supplied value in °F to °C. For now, do not worry about handling nonsensical input values; we will deal with such things in the next chapter.

8. Evaluate the following expressions: (a) `5 / 0`, (b) `0 / 0`, (c) `Infinity + Infinity`, (d) `Infinity - Infinity`, (e) `Infinity * Infinity`, (f) `Infinity / Infinity`, and (g) `Math.sqrt(Infinity)`. Do these results make sense? Why or why not?

9. Write a one-line script that alerts a greeting message in Armenian, Arabic, Hebrew, Hindi, Chinese, or any other language using a non-Latin script.

10. Give a JavaScript string for the formula $\forall P.\ P0 \wedge (\forall k.\ Pk \Rightarrow P(k+1)) \Rightarrow \forall n.\ Pn$. Show how it is represented in HTML as well.

11. Modify the die-rolling web page from this chapter to roll five dice instead of just one.

12. Repackage the die-rolling page from this chapter so that the HTML document and the script are in separate files.

13. Write an HTML document, with an embedded script, containing a single text field and buttons labeled `floor`, `ceil`, `sqrt`, `sin`, `cos`, `tan`, `abs`, and `log`. Clicking on any of the buttons should cause the corresponding mathematical operation to be applied to the value in the text field and to be displayed somewhere on the page. Because we have not covered a great deal of JavaScript yet, you can build this web page using the die-rolling page from this chapter as a guide, and write a separate "onclick" function for each button. Later in the text, we will show how to write such applications much more efficiently.

14. Evaluate the following expressions:

    (a) `"one two three".split(" ");`

    (b) `"abracadabra".split("a");`

15. Write a JavaScript expression that produces `true` if a string *s* contains a comma, and `false` otherwise.

16. Write a script that prompts for a string then alerts whether or not the entered string contains either a backslash character or a Telugu letter ddha (`U+0C22`).

17. Research the JavaScript `substring` operation. What happens if its first operand is larger than the length of the input string? What happens if the first operand is acceptable but the second operand is too large? What if one or the other operand is negative?

18. Let `p = { x: 1, y: [ 4, {z: 2} ] }`. What is `p.y[1]`?

19. What is the value of `({ x: 1, y: 2 }).y`? Would you ever write something like that?

20. What is the difference between `p["dog"]` and `p[dog]`? Write a script that illustrates this difference.

21. Explain the output of the script

```
var employee = {
    name: "Kaela",
    department: "Technology"
};
alert(employee.salary === undefined);
```

    in detail. Does this behavior make sense in light of the intended meaning of `null` and `undefined`?

22. In the script on page 73, replace the script with the following:

```
var style = document.getElementById("face").style;
setInterval(function () {
    style.left = Math.floor(Math.random() * 500) + 'px';
    style.top = Math.floor(Math.random() * 400) + 'px';
}, 2000);
```

Do you find this more or less readable? Why?

23. Draw a picture of the following object(s). Don't forget to show references as arrows.

```
{call: "mark", next: {call: "ready", next: {call: "set",
next: {call: "go", next: null}}}}
```

24. Draw a picture of the following object(s). Don't forget to show references as arrows. What do you think it means?

```
{op: "+", l: {op: "*", l: {op: "-", l: 3, r: 9}, r: 7},
r: {op: "/", l: 9, r: {op: "+", l: 8, r: 2}}}
```

25. Write a script that prompts for the abbreviation of a New England state and alerts the capital of that state. The first line should be:

```
var capitals = {ME: "Augusta", NH: "Concord", VT: "Montpelier",
    MA: "Boston", CT: "Hartford", RI: "Providence"};
```

For example, if your user inputs `NH`, your script should alert `Concord`.

26. Write variable declarations for the following:

- An employee whose name is María, salary is 1000 USD, hire date is 2008-01-05, and who does *not* have a supervisor.
- An array of the first 10 prime numbers.
- The song "Johnny Tarr" by Gaelic Storm from the album *Tree*. Gather as much information about the track as you can.

27. Draw a picture of the following object:

```
[42, true, null, NaN, "nil", {}, undefined]
```

28. Recall this little script from the chapter:

```
var mom = {name: "Ani"};
var dad = {name: "Sipho", spouse: mom};
mom.spouse = dad;
```

What is the value of `mom.spouse.spouse.spouse.spouse`?

29. Here is an attempt to recreate the objects in the previous exercise:

```
var mom = {name: "Ani", spouse: {name: "Sipho", spouse: mom}};
```

Assuming that the variable mom was not previously defined, draw a picture of the resulting object. Hint: You might want to use a shell or runner page for help.

30. Draw a picture of the objects created by the following:

```
var players = {name: "Moe"};
players.next = {name: "Larry"};
players.next.next = {name: "Curly", next: players};
```

Can you write a cleaner set of statements that constructs the same object network?

31. Write a script that objects for people named Ani, Sipho, Tuulia, ‘Aolani, Hiro, and Xue, such that:

- Tuulia is the mother of Sipho.
- Ani and Sipho are married.
- The children of Ani and Sipho are, in order, ‘Aolani, Hiro, and Xue.

Define each of the objects with as many of the following properties as you can fill in: `name`, `mother`, `father`, `spouse`, and `children`. The `children` property should have an array value.

32. Draw a picture of the array referred to by the variable `a` after executing the following:

```
var a = [1,2,3,4]; a.unshift(a.pop());
```

33. In mathematics, a *two-dimensional matrix* is an arrangement of values in rows and columns, such as:

$$\begin{pmatrix} 9 & 8 & 2 & -5 \\ \pi & 7 & 2.8 & 6 \\ -22 & 4 & 0 & 100 \end{pmatrix}$$

Matrices can be represented in JavaScript as arrays of arrays. Write a JavaScript array expression for the preceding matrix. Your array should have three elements, each of which is an array of four elements.

34. Let x stand for some completely arbitrary JavaScript expression. What can the expression `!!`x be used for?

35. Evaluate the following expressions and explain the results. Hint: You may want to review Section 3.8 on type conversion for help.

    (a) `5 < 10 < 20`

    (b) `-20 < -10 < -5`

36. Write a script that prompts for a string of text and alerts the string made up of this value appended to itself twice. For example, if the user inputs `"ho"`, your script should alert `"hohoho"`. If the user inputs `"888"`, your script should alert `"888888888"`.

37. Explain the result of `parseInt("250", 3)`.

38. Give the numeric value of each of the following expressions (write your answer in base-10): `791`, `0x2e5`, `2e5`, `0791`.

39. Write a script that prompts for a number in base-16 and alerts its value in base-10.

40. Determine the precedence of the `typeof` operator, relative to the addition, multiplication, and unary negation operators, by experimentation in the shell.

41. In mathematics, we are used to the fact that if $A = B$ and $B = C$ then $A = C$, a property known as the transitivity of equality. Because of implicit conversions, JavaScript's quirky `==` operator is not transitive. Demonstrate that transitivity does not hold. (Hint: Find a certain string A, a certain number B, and a certain string C for which A `==` B, B `==` C, but A `!=` C.) Is `===` transitive? Why or why not?

42. What JavaScript value should you assign to a variable `x` so that `typeof x === x` would be true?