CHAPTER 1

JavaScript

We'll begin our tour with JavaScript, because it is, by *some* measures, the most popular programming language in the world.

First appeared	1995
Creator	Brendan Eich
Notable versions	ES3 (1999) • ES5 (2009) • ES2015 (2015)
Recognized for	First-class functions, Weak typing, Prototypes
Notable uses	Web application clients, Asynchronous servers
Tags	Imperative, Functional, Dynamic, Prototypal
Six words or less	"The assembly language of the web"

JavaScript was designed and implemented in ten days in 1995 by Brendan Eich, then at Netscape Communications Corporation, with the goal of creating an amateur-friendly scripting language embedded into a web browser. The syntax of the language was strongly influenced by C, with curly braces, assignment statements, and the ubiquitous `if`, `while`, and `for` statements. Semantically, however, JavaScript and C are worlds apart. JavaScript's influence here was the lesser-known language Scheme. Functions are **first-class values**: they can be assigned to variables, passed to functions, and returned from functions.

The goal of allowing novice programmers to write small scripts in web page markup led to some well-loved design choices, including array (e.g., `[10, 20, 30]`) and object (e.g., `{x:3, y:5}`) literals. Yet the attempt to keep the language simple led to several notorious features as well. **Weak typing**, where expressions of the wrong type are automatically coerced to "something that works," and **automatic semicolon insertion**, where the language will figure out where your statements begin and end when you are not explicit, save typing but sometimes produce utterly surprising behavior. Douglas Crockford [20] has catalogued these and a number of other "Bad Parts" and "Awful Parts," while at the same time praising JavaScript as "[having] some extraordinarily good parts. In JavaScript there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders." [20, p. 2]

Though originally targeted to beginners, the language has grown a great deal over time and has been used in thousands of successful, sophisticated applications. Every major web browser comes with a JavaScript engine. Running applications written in languages other

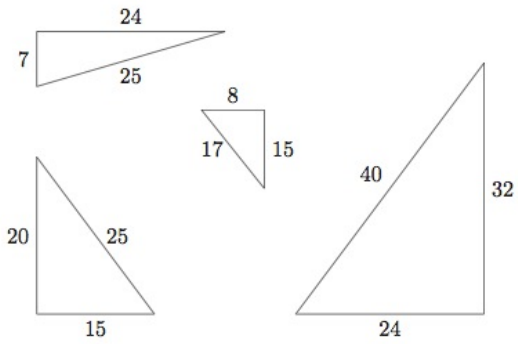


Figure 1.1 Integer right-triangle lengths

than JavaScript in a browser (including graphics-intensive applications written in C++ using the Unreal Engine [30]) is often done by first translating, or compiling, to JavaScript. But JavaScript's success is not limited to the browser; the language powers server-based applications supporting thousands of concurrent users.

As the first language on our tour, we'll use JavaScript to introduce several aspects of functions: argument passing, scope, the handling of free variables, closures, and anonymous and higher-order functions. We'll show how prototypes enable us to create sets of similar objects, and look at one of JavaScript's more interesting features: the all-purpose `this`-expression, which allows code to behave in certain ways based on context. We'll continue with a brief look at the concept of scope. We'll close the chapter with a short overview of asynchronous programming, one of many approaches to concurrent programming we'll encounter in this book.

1.1 HELLO JAVASCRIPT

Let's get a feel for JavaScript by generating some lucky numbers:

```
for (let c = 1; c <= 40; c++) {
  for (let b = 1; b < c; b++) {
    for (let a = 1; a < b; a++) {
      if (a * a + b * b === c * c) {
        console.log(`${a}, ${b}, ${c}`);
      }
    }
  }
}
```

We're terribly sorry for the flashback to high school math, but this code works pretty well as an opener, and is certainly more interesting than "Hello, world." It outputs all possible right-triangle measurements with integer values up to size 40 (see Figure 1.1 for examples) as follows:

```
3, 4, 5
6, 8, 10
5, 12, 13
9, 12, 15
8, 15, 17
12, 16, 20
15, 20, 25
7, 24, 25
10, 24, 26
20, 21, 29
18, 24, 30
16, 30, 34
21, 28, 35
12, 35, 37
15, 36, 39
24, 32, 40
```

As mentioned in the preface of this book, we're expecting you to have programming experience, so you can probably figure out the for-loops and the if-statement. Note the spelling of the equality operator: `x === y` is true iff x and y have the same value and the same type. We don't use JavaScript's `==` operator, as it only computes whether two objects are *similar* to, rather than equal to, each other, and sometimes behaves unexpectedly (e.g., `null == undefined` and `false == " \t "`). Within backquote-delimited strings, the construct `${`e`}` **interpolates** the value of expression e into the string.

Our second example writes the permutations of its **command line argument** to standard output. The language does not define how to access the command line, so we'll have to choose a specific implementation. We'll use the popular Node.js [70]:

```
function generatePermutations(a, n) {
  if (n === 0) {
    console.log(a.join(''));
  } else {
    for (let i = 0; i < n; i++) {
      generatePermutations(a, n - 1);
      const j = n % 2 === 0 ? 0 : i;
      [a[j], a[n]] = [a[n], a[j]];
    }
    generatePermutations(a, n - 1);
  }
}

if (process.argv.length !== 3) {
  console.error('Exactly one argument is required');
  process.exit(1);
}
const word = process.argv[2];
generatePermutations(word.split(''), word.length - 1);
```

Let's run this script:[1]

```
$ node --use-strict anagrams.js rat
rat
art
tra
rta
atr
tar
```

The script first checks that we have passed a single argument, and if not, terminates with an error message (written to standard error, *not* standard output) and non-zero exit code. Node's built-in variable `process.argv` contains the command line tokens invoking our script (excluding options), in our case:

```
process.argv === ['node', 'anagrams.js', 'rat']
```

A value of 3, therefore, indicates that one argument was passed to the script. Next, we call the recursive `generatePermutations` function of two arguments. We won't describe how the algorithm works; it's *Heap's algorithm*, which you can look up at Wikipedia. We will, however, note the script's use of `split` and `join`. JavaScript strings are **immutable**: you cannot make a string longer or shorter or change any of its constituent characters. Implementing Heap's algorithm requires us to split the string into an *array* of its characters, as arrays *are* mutable. We repeatedly rearrange the array in place through a series of swaps, each time applying the `join` operation to get a new string for output.

Let's move on to a more complex example. We'll read text from standard input and produce a report with the number of times each word appears. For now, words will consist only of the Latin letters A-Z and apostrophes; we'll remedy this deficiency later in the chapter. We'll again use Node.js, as we need a JavaScript runtime providing access to standard input:

```
const readline = require('readline');
const reader = readline.createInterface(process.stdin, null);
const counts = new Map();

reader.on('line', line => {
  for (let word of line.toLowerCase().match(/[a-z']+/g) || []) {
    counts.set(word, (counts.get(word) || 0) + 1);
  }
}).on('close', () => {
  for (let word of Array.from(counts.keys()).sort()) {
    console.log(`${word} ${counts.get(word)}`);
  }
});
```

To see the script in action, download, for fun, the plain text version of the complete works of Robert Burns from Project Gutenberg (`http://www.gutenberg.org/cache/epub/18500/pg18500.txt`) and call the file *burns.txt*. Store the script in *wordcount.js* and run:

[1] JavaScript scripts, and even certain portions of scripts, can be run in either strict mode or non-strict mode. The details of the two modes would fill many pages. For simplicity, all of the examples in this chapter assume strict mode, which can be triggered by invoking Node with the `-use-strict` option.

```
node --use-strict wordcount.js < burns.txt
```

This script uses Node's built-in `readline` module, and sets up a reader to read line-by-line from standard input. It then creates an empty map that we will fill with words and their counts. Next, the script ensures that whenever a line is ready (i.e., the system has fired a `line` event), it will be lowercased, broken up into words, and the counts for each word incremented. The script extracts from the lowercased line by matching against a **regular expression**[2] defining a word as a sequence of one or more letters in the range `a-z` and apostrophes. Finally, it arranges that when input has been fully read (via a `close` event), the script will write out each word, and its count, in sorted order.

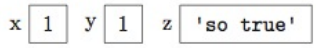
1.2 THE BASICS

A JavaScript program is made up of scripts and modules, each containing a sequence of statements and function declarations. Statements include variable declarations, assignments, function calls, conditionals and loops, among others. Values have one of exactly 7 types:

- The type containing the sole value `undefined`.
- The type containing the sole value `null`.
- **Boolean**, containing the two values `true` and `false`.
- **Number**, the type of all numbers, including -98.88, 22.7×10^{100}, `Infinity`, `-Infinity`, and, strangely enough, `NaN`, the number meaning "not a number."
- **String**, roughly, the type of character sequences, but technically the type of sequences of UTF-16 code points.[3] String literals are delimited by either single quotes, double quotes, or backquotes, with the latter allowed to span lines and contain interpolated expressions.
- **Symbol**, the type of symbols (not covered in this chapter).
- **Object**, the type of all other values, including arrays and functions. Objects have named properties each holding a value, for example `{x: 3, y: 5}`. Properties of an array include `0`, `1`, `2`, and so on, and `length`.

The first six types are **primitive types**; Object is a **reference type**. The difference can be explained by picturing variables as boxes containing values. The *declarations* `let x = 1; let y = true; let z = 'so ' + y` create variables with initial values as follows:

After a variable is declared, an **assignment** puts a new value in its box; for example, `y = x` will *copy* the value currently in `x` into `y` (overwriting the contents of `y`):



Values of a primitive type are written directly inside the variable boxes, but object values are actually **references** to entities holding the object properties. This is best explained by analyzing the following script, and its visualization in Figure 1.2:

[2] Complete details regarding regular expressions are beyond the scope of this text.
[3] Details of code points can be found in Appendix B

```
const a = {x: 3, y: 5};  // creates an object
const b = a.y;           // simply puts 5 into b
const c = null;          // simply puts null into c
const d = {x: 3, y: 5};  // creates an object
const e = d;             // does not create an object
```

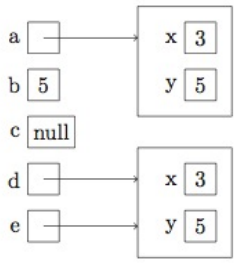


Figure 1.2 JavaScript Primitives and References

References allow multiple variables to refer to the same object. In our figure, `d.x` and `e.x` are **aliases** of each other—an assignment to one updates the other. References lead to multiple interpretations of what it means to copy: when nested objects exist, will copying only the references suffice (a **shallow copy**), or must *all* of the data be copied (a **deep copy**)? To perform a shallow copy, iterate through the properties of an object and assign their values to properties in the copy, or, for arrays, use JavaScript's `slice` method (see Figure 1.3):

```
const a = [{x:0, y:0}, {x:3, y:0}, {x: 3, y:4}];

const b = a;          // copies the reference, nothing more
const c = a.slice();  // makes a SHALLOW COPY of array elements
```

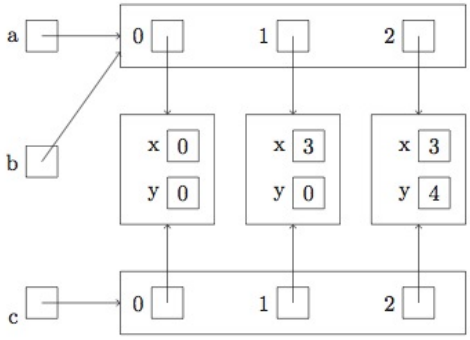


Figure 1.3 A Shallow Copy

To make a deep copy, iterate through the components of an object, copying primitives and recursively creating deep copies of objects.

JavaScript is a **weakly-typed language**, because more often than not, values of one type can appear where values of other types are expected.[4] For example:

- In `if` and `while` statements expecting a boolean condition, any value can appear. `0`, `null`, `undefined`, `false`, `NaN`, and the empty string act as false and are called **falsy**; all other values act as true and are called **truthy**.
- When a string is expected, `undefined` acts as `"undefined"`, `null` acts as `"null"`, `false` acts as `"false"`, `3` acts as `"3"`, and so on. To use an object x in a string context, JavaScript evaluates x`.toString()`.
- When a number is expected, `undefined` acts as `NaN`, `null` as `0`, `false` as `0`, `true` as `1`, and strings act as the number they "look like" or `NaN`. To use an object x in a numeric context, JavaScript evaluates x`.valueOf()`.

Table 1.1 provides concrete examples.

Value	as Boolean	as String	as Number
`undefined`	`false`	`'undefined'`	`NaN`
`null`	`false`	`'null'`	`0`
`false`	`false`	`'false'`	`0`
`true`	`true`	`'true'`	`1`
`0`	`false`	`'0'`	`0`
`858`	`true`	`'858'`	`858`
`NaN`	`false`	`'NaN'`	`NaN`
`'0'`	`true`	`'0'`	`0`
`'858'`	`true`	`'858'`	`858`
`''`	`false`	`''`	`0`
`'dog'`	`true`	`'dog'`	`NaN`
`Symbol('dog')`	`true`	`'Symbol(dog)'`	*throws* `TypeError`
any object x	`true`	*result of* x`.toString()`	*result of* x`.valueOf()`

Table 1.1 JavaScript Automatic Type Conversions

Implicit type conversion, also known as **coercion** tends to be the rule in JavaScript. `TypeError`s are very rare, thrown when using a symbol as a number, `null` or `undefined` as an object, or trying to call a non-function.

JavaScript gives us a few options for specifying function values, including the arrow (`=>`), the construct `function (`*params*`) {` *body* `}`, and the function declaration. Functions can be called via the name bound to the function in its declaration, the variable to which the function value was assigned, or even **anonymously**. Functions that accept functions as parameters or return functions are called **higher-order functions**. Higher-order functions facilitate a style of coding—called **functional programming**—in which function composition replaces assignment statements and explicit loops. Examples follow:[5]

[4] Contrast this with a strongly-typed language, in which using values of the wrong type more often than not generates an error.

[5] In this book, we'll be illustrating many behaviors via runnable scripts featuring assertions. Assertions either succeed quietly or fail by throwing an exception. Node.js provides a number of assertions in its `assert` module. In particular: `assert(`e`)` tests whether e is truthy, `assert.strictEqual(`$e1$`,`$e2$`)` tests whether $e1$ `===` $e2$, `assert.deepStrictEqual(`$e1$`,`$e2$`)` tests whether objects $e1$ and $e2$ have the same structure with equal values throughout, and `assert.throws(`f`)` tests whether function f will throw an exception when called.

```
const assert = require('assert');

// Function values can use `=>` or `function`
const square = x => x * x;
const odd = x => Math.abs(x % 2) === 1;
const lessThanTen = function (x) {return x < 10};
const twice = (f, x) => f(f(x));

// An anonymous function call
assert((x => x + 5)(10) === 15);

// We can pass function values to other functions
assert(twice(square, -3) === 81);
assert(twice(x => x + 1, 5) === 7);

// We can create and return new functions on the fly
function compose(f, g) {
  return x => f(g(x));
}
const isOddWhenSquared = compose(odd, square);
assert(isOddWhenSquared(7));
assert(!isOddWhenSquared(0));

// Array functions often take the place of loops
const a = [9, 7, 4, -1, 8];
assert(!a.every(odd));
assert(a.some(odd));
assert(a.every(lessThanTen));
assert.deepStrictEqual(a.filter(odd), [9, 7, -1]);
assert.deepStrictEqual(a.map(square), [81, 49, 16, 1, 64]);
```

In JavaScript, **arguments** are fully evaluated before the call and their values are assigned (copied) to the **parameters** left-to-right. If you pass too many arguments, the extras are ignored; pass too few and the extra parameters begin as `undefined`. You can, however, mark your final parameter with `...` to pack extra arguments into an array. You may also use `...` on the argument side, to unpack an array to pass into multiple parameters.

```
const assert = require('assert');

function f(x, y) {return [x, y]}
function g(x, ...y) {return [x, y]}

assert.deepStrictEqual(f(1), [1, undefined]);     // too few args
assert.deepStrictEqual(f(1, 2, 3), [1, 2]);       // too many args
assert.deepStrictEqual(g(1, 2, 3), [1, [2, 3]]);  // args packed
const args = [1, 2]
assert.deepStrictEqual(f(...args), [1, 2]);       // args unpacked
```

Each function call allocates a **frame**, or **activation record**, to hold parameters and **local variables** for this call. Local variables are those **declared** with `var`, `let`, or `const` inside the

function. Let's illustrate how function calls are implemented by revisiting the permutations script from Section 1.1. Consider running the script with argument `rat`. The script calls `generatePermutations(['r','a','t'], 2)`, generating a new frame with local variable `i`. This activation makes a call to `generatePermutations(a,1)`, which calls `generatePermutations(a,0)`. Figure 1.4 shows a snapshot of program execution at this point, at the beginning of the third activation. The figure shows three important aspects of JavaScript function implementation: (1) variables live inside frames, while objects live outside frames, (2) recursive solutions work because each activation gets its own copies of local variable and parameters, and (3) passing objects is "cheap," as only references, and not object internals, are copied.

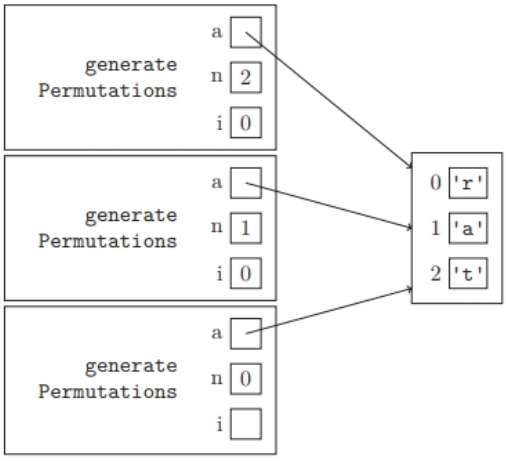


Figure 1.4 Function Call Execution Snapshot

Because parameters are always freshly allocated at call time, they are distinct variables from the arguments, so changing a parameter will not affect its corresponding argument. But keep in mind that since object values are just references, you can change the *properties* of a passed object through the parameter. Let's see how this works with a concrete example:

```
const assert = require('assert');

const x = [1,2,3];
const y = [4,5,6];

function f(a, b) {
  a = 300;                                // Change *parameter*
  assert.deepStrictEqual(x, [1,2,3]);     // Argument still intact!
  b[1] = 400;                             // Change *property*
  assert.deepStrictEqual(y, [4,400,6]); // See the change!
}

f(x, y);
```

Variables declared outside of any function or module are called **global variables** and are implemented in JavaScript as properties of the **global object**. Strictly speaking, properties are not variables at all: they live inside an object, not a frame! The global object contains dozens of properties including `isFinite`, `String`, `Object`, and `Date`, as well as a property that references the global object itself, called `global` or `window`, depending on the environment. And here's an interesting language design choice: referencing an undeclared variable throws a `ReferenceError`, while referencing a missing property produces `undefined`.

```
const assert = require('assert');

const p = {x: 3, y: 5}
assert(p.z === undefined);                    // There's no p.z
assert.throws(() => z, ReferenceError);       // No variable z
```

1.3 CLOSURES

Variables used but not declared inside the function are called **free variables**. All of the functions we have seen so far operate only on parameters and local variables, so let's see how JavaScript handles free variables.

```
const x = 'OUTER';
function second() {console.log(x);}
function first() {const x = 'FIRST'; second();}
first();
```

The variable x within `second` is free. Does x take on the value from its caller, function `first`? Or from the outer x? Languages that use caller's values for free variables are **dynamically scoped**; those that look outward to textually enclosing regions are **statically scoped**. JavaScript is statically scoped.

Things become even more interesting when functions are nested inside other functions. Consider this script:

```
function second(f) {
  const name = 'new';
  f();
}

function first() {
  const name = 'old';
  const printName = () => console.log(name);
  second(printName);
}

first();
```

The function assigned to `printName` in `first` has a free variable; it is passed to `second` (as f) and then executed, logging the value of `name`. When the function was created, it sees `name` defined within `first`, but when called (as f), might it see the `name` variable in

`second`? If a system binds the free variables of a passed function after passing, we speak of **shallow binding**; if bound where the function is defined, we have **deep binding**.

What does JavaScript do? When a nested function with free variables is sent outside its environment, either by being passed to or returned from another function, the function carries the bindings of those variables from the enclosing environment *of its definition* with it. These bindings "close over" the inner function, so the function, together with its bindings, is called a **lexical closure**, or **closure** for short.

Closures can be used to make **generators**.[6] A generator function produces a "next" value each time it is called. A generator for a sequence of squares would produce 0 on its first call, 1 on its second, then 4, 9, 16, and so on. A function that increments a global variable then returns its square would be **insecure** because other parts of the code could change the global variable, disrupting future calls to the generator! Fortunately, we can use the fact that a function's local variables are completely hidden from the outside:

```
const nextSquare = (() => {
  let previous = -1;
  return () => {
    previous++;
    return previous * previous;
  }
})();

const assert = require('assert');
assert(nextSquare() === 0);
assert(nextSquare() === 1);
assert(nextSquare() === 4);
```

The value assigned to `nextSquare` is the result of calling an anonymous function; we call the right-hand side of the assignment an **immediately invoked function expression**, or IIFE. The call returns a closure. The variable that holds the number to be squared is local to the enclosing function. The generator (`nextSquare`) can see this value, but no other parts of the code can. The generator is secure.

1.4 METHODS

An object can have properties whose values are functions:

```
const circle = {
  radius: 10,
  area: function () {return Math.PI * this.radius * this.radius},
  circumference: function () {return 2 * Math.PI * this.radius},
  expand: function (scale) {this.radius *= scale}
};
```

When we call a function via property access notation (e.g., `circle.area()`), we say the function is a **method** and the object is the **receiver**. Note that we've used the long syn-

[6] Generators are more commonly created with the `yield` statement, but we're using a low-level approach here simply to illustrate closures in action.

tax for function values: if we define the method value with the `function ( params ) { body }` syntax, the special expression `this` refers to the receiver. Functions defined with `( params ) => { body }` do not get a special `this`. You may wish to adopt the convention of using the `function` syntax for methods and the arrow notation in all other cases. If you prefer, there's a shorthand notation for the `function` syntax inside of object literals:

```
const circle = {
  radius: 10,
  area() {return Math.PI * this.radius * this.radius},
  circumference() {return 2 * Math.PI * this.radius},
  expand(scale) {this.radius *= scale}
};
```

The purpose of the special `this` expression is to allow *context-dependent code*. In the case of methods, `this` takes on the value of the method's receiver as determined at runtime. For example, if we copy a method defined in object A to object B and call the method through B, `this` will be B, not A. This **late binding** can be quite flexible, as we'll see in the next section.

JavaScript employs `this` in situations other than method calls. We can, for instance, force the value of `this` to take on the value of our choosing via `call`, `apply`, and `bind`:

```
function talkTo(message, suffix) {
  return message + ', ' + this.name + suffix;
}

const alice = {name: 'Alice', address: talkTo};
const bob = {name: 'Bob'};

const assert = require('assert');
assert(alice.address('Hello', '.') === 'Hello, Alice.');
assert(alice.address.call(bob, 'Yo', '!') === 'Yo, Bob!');
assert(alice.address.apply(bob, ['Bye', '...']) === 'Bye, Bob...');
assert(alice.address.bind(bob)('Right', '?') === 'Right, Bob?');
```

1.5 PROTOTYPES

Let's turn now from functions to objects. How do you efficiently create a number of similar objects? How do you define dozens, thousands, or millions of points, or of circles, or people, or votes, or airports, or web page index entries?

In JavaScript, we start with an initial (prototypical) object, then *derive* additional objects from it. These new objects have the original object as their **prototype**. What is a prototype? When we encounter the expression $q.x$, we look for an x property in q. If found, we produce the corresponding value; if not, we'll look in q's prototype (if it has one), and if necessary, in the prototype's prototype, and so on, until we find the property or reach the end of the "prototype chain." If no object on the chain has the property, the lookup produces `undefined`.

```
const unitCircle = {
  x: 0,
  y: 0,
  radius: 1,
  color: 'black',
  area() {return Math.PI * this.radius * this.radius},
  circumference() {return 2 * Math.PI * this.radius}
};

const c1 = Object.create(unitCircle);
c1.x = 3;
c1.color = 'green';

const c2 = Object.create(unitCircle);
c2.radius = 5;

const c3 = Object.create(unitCircle);

const assert = require('assert')
assert(c2.color === 'black' && c2.area() === 25 * Math.PI);
assert(c3.y === 0 && c3.area() === Math.PI);
```

The expression `Object.create(p)` creates a new object whose prototype is p. Our script creates a black circle of radius 1, centered at the origin, as the prototype of three other circles. Because of the way JavaScript **delegates** property lookup, we have `c1.x === 3` and `c1.color === 'green'` (obviously), as well as `c1.y === 0` and `c1.radius === 1`. In the object referenced by `c1`, `x` and `color` are called **own properties**, while `y` and `radius` are called **inherited properties**.

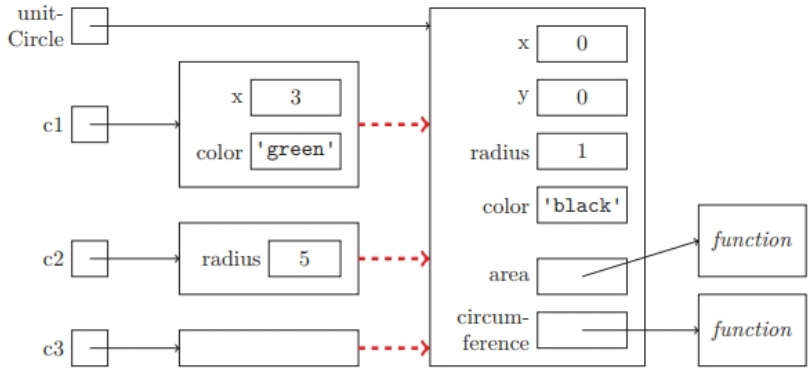


Figure 1.5 Objects sharing a prototype

Note that in each of the newly created objects, we store only those properties whose values *differ* from those in the prototypal circle. In particular, each circle inherits all of the methods from the prototype. Every circle computes its area and circumference the same way, so it would be wasteful to store copies of these functions in each circle.

Figure 1.5 shows the four circles and the prototype links to `unitCircle`. For simplicity, our diagram omits the prototype links from `unitCircle` to its prototype, and from the two functions to their prototype.[7]

In practice, programmers will want a function to construct each instance of a family of objects that each share a prototype. Interestingly, JavaScript provides a mechanism to do just that. We'll illustrate the technique with an example, and then discuss:

```
function Circle(centerX=0, centerY=0, radius=1, color='black') {
  this.x = centerX;
  this.y = centerY;
  this.radius = radius;
  this.color = color;
}

Circle.prototype.area = function () {
  return Math.PI * this.radius * this.radius;
};

Circle.prototype.circumference = function () {
  return 2 * Math.PI * this.radius;
};

const assert = require('assert');
const c = new Circle(1, 5);
assert.deepEqual(c, {x:1, y:5, radius:1, color:'black'})
assert(c.area() === Math.PI);
assert(c.circumference() === 2 * Math.PI);
assert(Object.getPrototypeOf(c) === Circle.prototype);
assert(c.constructor) === Circle;
assert(typeof(c) === 'object');
```

This simple looking script illustrates a lot of JavaScript magic. Every JavaScript function has two properties, `length` (the number of parameters) and `prototype`, the object that will be assigned as the prototype of all objects created by calling the function with the operator `new`. It's primed for you with a `constructor` property, referencing the function. This allows you to determine the function that created an object as a kind of run time "type check," since, as the last line of our example shows, `Circle` isn't a JavaScript type. That's a lot to take in, but studying Figure 1.6 may help!

The declaration `let c = new Circle(1, 5);` creates a new object with prototype `Circle.prototype`, passes it to the function `Circle` as `this`, and binds the now-initialized object to the variable *c*. The area and circumference methods are stored as properties in the prototype, just as in our earlier example. As an aside, we've taken the opportunity here to introduce **default parameter values**—parameters initialized to specified values, rather than `undefined`, when no argument is supplied.

Creating a constructor function to build **instances** of a user-defined type, and loading up the shared state and behavior into a common prototype occurs so often in JavaScript that there is a shorthand syntax for this pattern:

[7] One of the end-of-chapter exercises asks you to complete the figure.

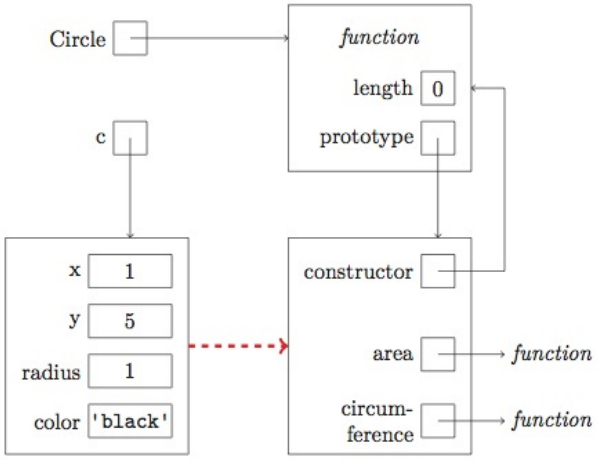


Figure 1.6 Construction of Objects using `new`

```
class Circle {
  constructor(centerX=0, centerY=0, radius=1, color='black') {
    this.x = centerX;
    this.y = centerY;
    this.radius = radius;
    this.color = color;
  }
  area() {return Math.PI * this.radius * this.radius;}
  circumference() {return 2 * Math.PI * this.radius;}
}

const assert = require('assert');
const c = new Circle(1, 5);
assert(c.circumference() === 2 * Math.PI);
assert(typeof Circle === 'function');
```

The last line of the script above shows us that the `class` keyword *does not* create a "class object": `Circle` is still a function! The class construct is **syntactic sugar**—a syntax that makes the standard form easier to read. It does no more and no less (in this case) than defining the function and assigning methods to the prototype.

1.6 SCOPE

A **binding** is an association of a name with an entity. The **scope of a binding** is the region of code where a particular binding is active. Let's take a look at two ways we can introduce bindings in JavaScript: `let` and `var` (`const` works here like `let`). Bindings introduced with `var` are scoped to the innermost function, and bindings introduced with `let` are scoped to the nearest block:

```
const assert = require('assert');

const a = 1, b = 2;

(function () {
  assert(a === undefined);   // the local `a` is in scope
  assert(b === 2);           // we see the outer `b`

  if (true) {
    var a = 100;             // scoped to whole function!
    let b = 200;             // scoped only inside this block
    const c = 300;             // scoped only inside this block
  }
  assert(a === 100);         // it's been initialized
  assert(b === 2);           // outer, because local used `let`

  assert.throws(() => c);    // there's no `c` out here at all
})()
```

Reading a `var`-declared variable in its scope but before its declaration produces `undefined`, as shown above. Reading a `let`-declared variable in its scope but before the `let` throws a `ReferenceError`.

Given that there are several other ways to create bindings (`const`, function declarations, class declarations, function parameters, etc.), the complete set of rules that determine scope would fill many pages. In general, we will not be detailing the complete scoping rules for any language in this book, though we will make time to show some of the more interesting design choices surrounding scope.

1.7 CALLBACKS AND PROMISES

Programmers often have to deal with uncertainties surrounding time. Users will, without any warning, click buttons, drag fingers across a surface, press and release keys, and move cursors in and out of regions on a display. Reading and writing files and databases, and exchanging information with programs running on different machines may take several seconds to a few minutes or more to complete. When an application stops and waits for external operations to complete, we call its behavior **synchronous**; if it can continue to do useful work until the long running operation finishes, we have **asynchronous** behavior.

An asynchronous architecture consists of code for *firing* and *responding to* **events**. Some events originate outside the program (button clicks, a cursor entering a canvas, data becoming ready from a file read) and some from **event emitters** that you write yourself. In either case, events are added to a task queue which the JavaScript engine repeatedly pulls from. An interactive application contains instructions saying "When event e is pulled from the queue, call function f with the data provided by e". Let's see how this looks in a browser. The following script creates a little canvas you can sketch in:[8]

[8] To run this script in a browser, save it in the file *sketch.js*, Create a new file, *sketch.html*, with the content `<script src="sketch.js"></script>`, and finally, open the HTML file in the browser.

```
window.addEventListener('load', e => {
  const canvas = document.createElement('canvas');
  const ctx = canvas.getContext('2d');
  let drawing = false;
  canvas.style.border = '2px solid purple';
  canvas.addEventListener('mousedown', e => {
    drawing = true;
    ctx.moveTo(e.clientX, e.clientY);
  });
  canvas.addEventListener('mousemove', e => {
    if (drawing) {
      ctx.lineTo(e.clientX, e.clientY);
      ctx.stroke();
    }
  });
  const stopDrawing = e => {drawing = false};
  canvas.addEventListener('mouseup', stopDrawing);
  canvas.addEventListener('mouseout', stopDrawing);
  document.body.appendChild(canvas);
});
```

The functions passed as the second argument of `addEventListener` are called **event handlers**, or **callbacks**, and the act of adding the listeners to the various objects known to the browser is called **registering** the callback. When an event is pulled from the queue, the browser passes an **event object**, containing data about the event, to your callback. Mouse events will contain the cursor position in the `clientX` and `clientY` properties; touch events for phones and tablets work in a similar fashion. Each of the callbacks is run to completion, one after the other.

To see how things work on the server side, let's extend our word count script from the beginning of the chapter. We'll allow words containing any letter from the Unicode character set, rather than limiting ourselves to the Basic Latin letters.[9]

```
const reader = require('readline').createInterface(process.stdin, null);
const XRegExp = require('xregexp').XRegExp;
const counts = new Map();

reader.on('line', line => {
  const wordPattern = XRegExp("[\\p{L}']+", 'g');
  for (let word of line.toLowerCase().match(wordPattern) || []) {
    counts.set(word, (counts.get(word) || 0) + 1);
  }
}).on('close', () => {
  for (let word of Array.from(counts.keys()).sort()) {
    console.log(`${word} ${counts.get(word)}`);
  }
});
```

[9] Full Unicode support for matching is not built-in to Node.js, you'll have to install an external module by invoking `npm install xregexp` on the command line.

Node's `process.stdin` is a file **stream** representing standard input. In Node, streams are event emitters; `stdin` will fire a `line` event when a line is ready to be read, and a `close` event when there is no more data to be read. We call the `on` method on the reader to register the callback to invoke when the event fires.

Node.js comes with a number of built-in modules that contain event emitters, such as `stream`, `fs` (for file system), `networking`, and `http`. Events are fired at all the expected times: when streams or files are opened or closed, when data from a file, stream, or socket becomes ready, or upon timeouts or errors.

As an alternative to writing asynchronous functions that take callbacks as parameters, you can write functions that return **promises** instead. A JavaScript promise is built from the `Promise` constructor of the standard library with a single argument, called the executor. The executor has two parameters, `resolve` and `reject`, running asynchronously, eventually calling `resolve` to indicate success or `reject` to indicate failure. You can build up a series of asynchronous calls by chaining promises with `then` (to capture successful resolutions) or `catch` (to capture rejections). The following example simulates a chain of three long running (three whole seconds each) asynchronous tasks. For simplicity, we illustrate potential "failure" only for the first task.

```
function initialize(configuration) {
  console.log('Initializing ' + configuration);
  return new Promise((resolve, reject) => {
    if (!configuration) {
      reject('Empty configuration');
    } else {
      setTimeout(() => resolve('the initialized data'), 3000);
    }
  });
}

function process(initialData) {
  console.log('Processing ' + initialData);
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve('the processed data'), 3000);
  });
}

function report(output) {
  console.log('Reporting ' + output);
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve('the reported data'), 3000);
  });
}

initialize('the configuration data')
  .then(text => process(text))
  .then(value => report(value))
  .then(value => console.log('Success: ' + value))
  .catch(reason => console.log('Error: ' + reason));
```

JavaScript's asynchronous callbacks and promises comprise only two ways to manage **concurrency**, the modeling and coordination of independent and distinct computing activities whose execution spans may overlap in time. We will see others throughout the book, including coroutines, threads, processes, actors, and functional reactive programming.

1.8 JAVASCRIPT WRAP UP

In this chapter we were introduced to JavaScript. We learned that:

- All modern web browsers run JavaScript; in addition, engines such as Node.js run JavaScript on the command line or server.
- JavaScript expressions manipulate values of exactly seven types: Undefined, Null, Boolean, Number, String, Symbol, and Object. The first six are primitive types; Object is a reference type. Arrays and functions are objects. Values of reference types are actually pointers, so assignment of these values creates aliasing, or sharing.
- JavaScript is weakly-typed, meaning that in most situations a value e of type t can be used in a context in which a value of type t' is expected. The runtime will find a coercion of e to a roughly-equivalent expression e' of type t'.
- Values that coerce to false are called *falsy*; all other values are called *truthy*. The only falsy values in JavaScript are `undefined`, `null`, `0`, `NaN`, the empty string, and the empty object.
- When calling functions, arguments are fully evaluated and then passed to parameters by copying values. Extra arguments are ignored; extra parameters are assigned `undefined`, or a default value if specified. Parameters and local variables live only during the function activation and are invisible to outer scopes.
- When functions with free variables are passed into, or copied into, different scopes, the free variables continue to refer to the variables in their originally enclosing scopes. Functions taking advantage of this ability are called closures.
- JavaScript is statically scoped and uses deep binding. It supports both function-scoped (`var`) and block-scoped (`let`, `const`) entities.
- Object properties are either *own properties* or *inherited properties*. Property lookup traverses the prototype chain, if required.
- The value of the expression `this` is context-dependent. It refers to (1) the global object when used in a global context, (2) the receiver of a method in a method call, provided the method is *not* defined with the fat arrow, (3) the newly created object when used with operator `new`, or (4) an object designated by the programmer in certain methods, such as `apply`, `call`, and `bind`.
- The keyword `class` provides sugar for defining a constructor function and methods that populate a prototype object.
- The design of JavaScript facilitates asynchronous, event-driven programming with callbacks. The standard library provides promises, which can be used instead of callbacks in many situations.

To continue your study of JavaScript beyond the introductory material of this chapter, you may wish to find and research the following: