UNIT-III

The Secret Life of Objects

The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

Joe Armstrong, *interviewed in Coders at Work*

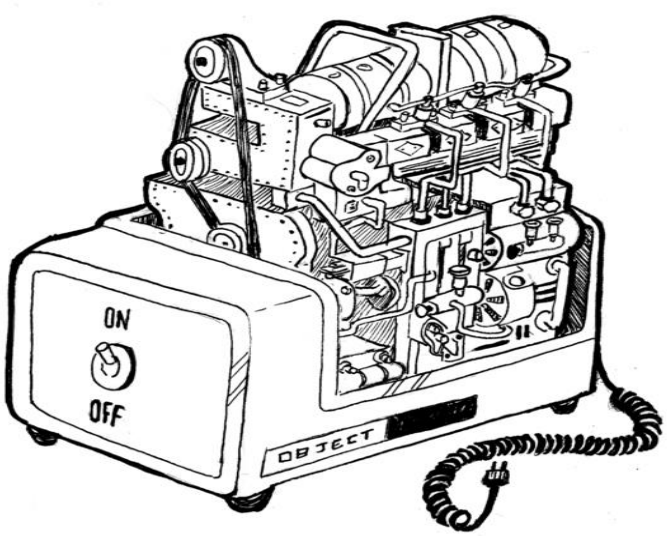
When a programmer says "object", this is a loaded term. In my profession, objects are a way of life, the subject of holy wars, and a beloved buzzword that still hasn't quite lost its power.

To an outsider, this is probably a little confusing. Let's start with a brief history of objects as a programming construct.

History

This story, like most programming stories, starts with the problem of complexity. One philosophy is that complexity can be made manageable by separating it into small compartments that are isolated from each other. These compartments have ended up with the name *objects*.

An object is a hard shell that hides the gooey complexity inside it and instead offers us a few knobs and connectors (such as methods) that present an *interface* through which the object is to be used. The idea is that the interface is relatively simple and all the complex things going on *inside* the object can be ignored when working with it.



As an example, you can imagine an object that provides an interface to an area on your screen. It provides a way to draw shapes or text onto this area but hides all the details of how these shapes are converted to the actual pixels that make up the screen. You'd have a set of methods—for example, drawCircle—and those are the only things you need to know in order to use such an object.

These ideas were initially worked out in the 1970s and 1980s and, in the 1990s, were carried up by a huge wave of hype—the object-oriented programming revolution. Suddenly, there was a large tribe of people declaring that objects were the *right* way to program—and that anything that did not involve objects was outdated nonsense.

That kind of zealotry always produces a lot of impractical silliness, and there has been a sort of counter-revolution since then. In some circles, objects have a rather bad reputation nowadays.

I prefer to look at the issue from a practical, rather than ideological, angle. There are several useful concepts, most importantly that of *encapsulation* (distinguishing between internal complexity and external interface), that the object-oriented culture has popularized. These are worth studying.

This chapter describes JavaScript's rather eccentric take on objects and the way they relate to some classical object-oriented techniques.

Methods

Methods are simply properties that hold function values. This is a simple method:

```
edit & run code by clicking it
var rabbit = {};
rabbit.speak = function(line) {
  console.log("The rabbit says '" + line + "'");
};

rabbit.speak("I'm alive.");
// → The rabbit says 'I'm alive.'
```

Usually a method needs to do something with the object it was called on. When a function is called as a method—looked up as a property and immediately called, as in object.method()—the special variable this in its body will point to the object that it was called on.

```
function speak(line) {
  console.log("The " + this.type + " rabbit says '" +
```

```
            line + """);
}
var whiteRabbit = {type: "white", speak: speak};
var fatRabbit = {type: "fat", speak: speak};

whiteRabbit.speak("Oh my ears and whiskers, " +
                  "how late it's getting!");
// → The white rabbit says 'Oh my ears and whiskers, how
//   late it's getting!'
fatRabbit.speak("I could sure use a carrot right now.");
// → The fat rabbit says 'I could sure use a carrot
//   right now.'
```

The code uses the this keyword to output the type of rabbit that is speaking. Recall that the apply and bind methods both take a first argument that can be used to simulate method calls. This first argument is in fact used to give a value to this.

There is a method similar to apply, called call. It also calls the function it is a method of but takes its arguments normally, rather than as an array. Like apply and bind, call can be passed a specific this value.

```
speak.apply(fatRabbit, ["Burp!"]);
// → The fat rabbit says 'Burp!'
speak.call({type: "old"}, "Oh my.");
// → The old rabbit says 'Oh my.'
```

Prototypes

Watch closely.

```
var empty = {};
console.log(empty.toString);
// → function toString(){…}
console.log(empty.toString());
// → [object Object]
```

I just pulled a property out of an empty object. Magic!

Well, not really. I have simply been withholding information about the way JavaScript objects work. In addition to their set of properties, almost all objects also have a *prototype*. A prototype is another object that

is used as a fallback source of properties. When an object gets a request for a property that it does not have, its prototype will be searched for the property, then the prototype's prototype, and so on.

So who is the prototype of that empty object? It is the great ancestral prototype, the entity behind almost all objects, Object.prototype.

```
console.log(Object.getPrototypeOf({}) ==
        Object.prototype);
// → true
console.log(Object.getPrototypeOf(Object.prototype));
// → null
```

As you might expect, the Object.getPrototypeOf function returns the prototype of an object.

The prototype relations of JavaScript objects form a tree-shaped structure, and at the root of this structure sits Object.prototype. It provides a few methods that show up in all objects, such as toString, which converts an object to a string representation.

Many objects don't directly have Object.prototype as their prototype, but instead have another object, which provides its own default properties. Functions derive from Function.prototype, and arrays derive from Array.prototype.

```
console.log(Object.getPrototypeOf(isNaN) ==
        Function.prototype);
// → true
console.log(Object.getPrototypeOf([]) ==
        Array.prototype);
// → true
```

Such a prototype object will itself have a prototype, often Object.prototype, so that it still indirectly provides methods like toString.

The Object.getPrototypeOf function obviously returns the prototype of an object. You can use Object.create to create an object with a specific prototype.

```
var protoRabbit = {
  speak: function(line) {
    console.log("The " + this.type + " rabbit says '" +
                line + "'");
```

```
  }
};
var killerRabbit = Object.create(protoRabbit);
killerRabbit.type = "killer";
killerRabbit.speak("SKREEEE!");
// → The killer rabbit says 'SKREEEE!'
```

The “proto” rabbit acts as a container for the properties that are shared by all rabbits. An individual rabbit object, like the killer rabbit, contains properties that apply only to itself—in this case its type—and derives shared properties from its prototype.

Constructors

A more convenient way to create objects that derive from some shared prototype is to use a *constructor*. In JavaScript, calling a function with the new keyword in front of it causes it to be treated as a constructor. The constructor will have its this variable bound to a fresh object, and unless it explicitly returns another object value, this new object will be returned from the call.

An object created with new is said to be an *instance* of its constructor.

Here is a simple constructor for rabbits. It is a convention to capitalize the names of constructors so that they are easily distinguished from other functions.

```
function Rabbit(type) {
  this.type = type;
}

var killerRabbit = new Rabbit("killer");
var blackRabbit = new Rabbit("black");
console.log(blackRabbit.type);
// → black
```

Constructors (in fact, all functions) automatically get a property named prototype, which by default holds a plain, empty object that derives from Object.prototype. Every instance created with this constructor will have this object as its prototype. So to add a speak method to rabbits created with the Rabbit constructor, we can simply do this:

```
Rabbit.prototype.speak = function(line) {
  console.log("The " + this.type + " rabbit says '" +
```

```
        line + """);
};
blackRabbit.speak("Doom...");
// → The black rabbit says 'Doom...'
```

It is important to note the distinction between the way a prototype is associated with a constructor (through its prototype property) and the way objects *have* a prototype (which can be retrieved with Object.getPrototypeOf). The actual prototype of a constructor is Function.prototype since constructors are functions. Its prototype *property* will be the prototype of instances created through it but is not its *own* prototype.

Overriding derived properties

When you add a property to an object, whether it is present in the prototype or not, the property is added to the object *itself*, which will henceforth have it as its own property. If there *is* a property by the same name in the prototype, this property will no longer affect the object. The prototype itself is not changed.

```
Rabbit.prototype.teeth = "small";
console.log(killerRabbit.teeth);
// → small
killerRabbit.teeth = "long, sharp, and bloody";
console.log(killerRabbit.teeth);
// → long, sharp, and bloody
console.log(blackRabbit.teeth);
// → small
console.log(Rabbit.prototype.teeth);
// → small
```

The following diagram sketches the situation after this code has run. The Rabbit and Object prototypes lie behind killerRabbit as a kind of backdrop, where properties that are not found in the object itself can be looked up.

Overriding properties that exist in a prototype is often a useful thing to do. As the rabbit teeth example shows, it can be used to express exceptional properties in instances of a more generic class of objects, while letting the nonexceptional objects simply take a standard value from their prototype.

It is also used to give the standard function and array prototypes a different toString method than the basic object prototype.

```
console.log(Array.prototype.toString ==
        Object.prototype.toString);
// → false
console.log([1, 2].toString());
// → 1,2
```

Calling toString on an array gives a result similar to calling .join(",") on it—it puts commas between the values in the array. Directly calling Object.prototype.toString with an array produces a different string. That function doesn't know about arrays, so it simply puts the word "object" and the name of the type between square brackets.

```
console.log(Object.prototype.toString.call([1, 2]));
// → [object Array]
```

Prototype interference

A prototype can be used at any time to add new properties and methods to all objects based on it. For example, it might become necessary for our rabbits to dance.

```
Rabbit.prototype.dance = function() {
  console.log("The " + this.type + " rabbit dances a jig.");
};
killerRabbit.dance();
// → The killer rabbit dances a jig.
```

That's convenient. But there are situations where it causes problems. In previous chapters, we used an object as a way to associate values with names by creating properties for the names and giving them the corresponding value as their value. Here's an example from Chapter 4:

```
var map = {};
function storePhi(event, phi) {
  map[event] = phi;
}

storePhi("pizza", 0.069);
storePhi("touched tree", -0.081);
```

We can iterate over all phi values in the object using a for/in loop and test whether a name is in there using the regular in operator. But unfortunately, the object's prototype gets in the way.

```
Object.prototype.nonsense = "hi";
for (var name in map)
  console.log(name);
// → pizza
// → touched tree
// → nonsense
console.log("nonsense" in map);
// → true
console.log("toString" in map);
// → true

// Delete the problematic property again
delete Object.prototype.nonsense;
```

That's all wrong. There is no event called "nonsense" in our data set. And there *definitely* is no event called "toString".

Oddly, toString did not show up in the for/in loop, but the in operator did return true for it. This is because JavaScript distinguishes between *enumerable* and *nonenumerable* properties.

All properties that we create by simply assigning to them are enumerable. The standard properties in Object.prototype are all nonenumerable, which is why they do not show up in such a for/in loop.

It is possible to define our own nonenumerable properties by using the Object.defineProperty function, which allows us to control the type of property we are creating.

```
Object.defineProperty(Object.prototype, "hiddenNonsense",
                      {enumerable: false, value: "hi"});
for (var name in map)
  console.log(name);
// → pizza
// → touched tree
console.log(map.hiddenNonsense);
// → hi
```

So now the property is there, but it won't show up in a loop. That's good. But we still have the problem with the regular in operator claiming that the Object.prototype properties exist in our object. For that, we can use the object's hasOwnProperty method.

```
console.log(map.hasOwnProperty("toString"));
// → false
```

This method tells us whether the object *itself* has the property, without looking at its prototypes. This is often a more useful piece of information than what the in operator gives us.

When you are worried that someone (some other code you loaded into your program) might have messed with the base object prototype, I recommend you write your for/in loops like this:

```
for (var name in map) {
  if (map.hasOwnProperty(name)) {
    // ... this is an own property
  }
}
```

Prototype-less objects

But the rabbit hole doesn't end there. What if someone registered the name hasOwnProperty in our map object and set it to the value 42? Now the call to map.hasOwnProperty will try to call the local property, which holds a number, not a function.

In such a case, prototypes just get in the way, and we would actually prefer to have objects without prototypes. We saw the Object.create function, which allows us to create an object with a specific prototype. You are allowed to pass null as the prototype to create a fresh object with no prototype. For objects like map, where the properties could be anything, this is exactly what we want.

```
var map = Object.create(null);
map["pizza"] = 0.069;
console.log("toString" in map);
// → false
console.log("pizza" in map);
// → true
```

Much better! We no longer need the hasOwnProperty kludge because all the properties the object has are its own properties. Now we can safely use for/in loops, no matter what people have been doing to Object.prototype.

Polymorphism

When you call the String function, which converts a value to a string, on an object, it will call the toString method on that object to try to create a meaningful string to return. I mentioned that some of the standard prototypes define their own version of toString so they can create a string that contains more useful information than "[object Object]".

This is a simple instance of a powerful idea. When a piece of code is written to work with objects that have a certain interface—in this case, a toString method—any kind of object that happens to support this interface can be plugged into the code, and it will just work.

This technique is called *polymorphism*—though no actual shape-shifting is involved. Polymorphic code can work with values of different shapes, as long as they support the interface it expects.

Laying out a table

I am going to work through a slightly more involved example in an attempt to give you a better idea what polymorphism, as well as object-oriented programming in general, looks like. The project is this: we will write a program that, given an array of arrays of table cells, builds up a string that contains a nicely laid out table—meaning that the columns are straight and the rows are aligned. Something like this:

```
name         height country
------------ ------ -------------
Kilimanjaro    5895 Tanzania
Everest        8848 Nepal
Mount Fuji     3776 Japan
Mont Blanc     4808 Italy/France
Vaalserberg     323 Netherlands
Denali         6168 United States
Popocatepetl   5465 Mexico
```

The way our table-building system will work is that the builder function will ask each cell how wide and high it wants to be and then use this information to determine the width of the columns and the height of the rows. The builder function will then ask the cells to draw themselves at the correct size and assemble the results into a single string.

The layout program will communicate with the cell objects through a well-defined interface. That way, the types of cells that the program supports is not fixed in advance. We can add new cell styles later—for example, underlined cells for table headers—and if they support our interface, they will just work, without requiring changes to the layout program.

This is the interface:

- minHeight() returns a number indicating the minimum height this cell requires (in lines).
- minWidth() returns a number indicating this cell's minimum width (in characters).
- draw(width, height) returns an array of length height, which contains a series of strings that are each width characters wide. This represents the content of the cell.

I'm going to make heavy use of higher-order array methods in this example since it lends itself well to that approach.

The first part of the program computes arrays of minimum column widths and row heights for a grid of cells. The rows variable will hold an array of arrays, with each inner array representing a row of cells.

```
function rowHeights(rows) {
  return rows.map(function(row) {
    return row.reduce(function(max, cell) {
      return Math.max(max, cell.minHeight());
    }, 0);
  });
}

function colWidths(rows) {
  return rows[0].map(function(_, i) {
    return rows.reduce(function(max, row) {
      return Math.max(max, row[i].minWidth());
    }, 0);
  });
}
```

Using a variable name starting with an underscore (_) or consisting entirely of a single underscore is a way to indicate (to human readers) that this argument is not going to be used.

The rowHeights function shouldn't be too hard to follow. It uses reduce to compute the maximum height of an array of cells and wraps that in map in order to do it for all rows in the rows array.

Things are slightly harder for the colWidths function because the outer array is an array of rows, not of columns. I have failed to mention so far that map (as well as forEach, filter, and similar array methods) passes a second argument to the function it is given: the index of the current element. By mapping over the elements of the first row and only using the mapping function's second argument, colWidths builds up an array with one element for every column index. The call to reduce runs over the outer rows array for each index and picks out the width of the widest cell at that index.

Here's the code to draw a table:

```
function drawTable(rows) {
  var heights = rowHeights(rows);
  var widths = colWidths(rows);

  function drawLine(blocks, lineNo) {
    return blocks.map(function(block) {
      return block[lineNo];
    }).join(" ");
  }

  function drawRow(row, rowNum) {
    var blocks = row.map(function(cell, colNum) {
      return cell.draw(widths[colNum], heights[rowNum]);
    });
    return blocks[0].map(function(_, lineNo) {
      return drawLine(blocks, lineNo);
    }).join("\n");
  }

  return rows.map(drawRow).join("\n");
}
```

The drawTable function uses the internal helper function drawRow to draw all rows and then joins them together with newline characters.

The drawRow function itself first converts the cell objects in the row to *blocks*, which are arrays of strings representing the content of the cells, split by line. A single cell containing simply the number 3776 might be represented by a single-element array like ["3776"], whereas an underlined cell might take up two lines and be represented by the array ["name", "----"].

The blocks for a row, which all have the same height, should appear next to each other in the final output. The second call to map in drawRow builds up this output line by line by mapping over the lines in the leftmost block and, for each of those, collecting a line that spans the full width of the table. These lines are then joined with newline characters to provide the whole row as drawRow's return value.

The function drawLine extracts lines that should appear next to each other from an array of blocks and joins them with a space character to create a one-character gap between the table's columns.

Now let's write a constructor for cells that contain text, which implements the interface for table cells. The constructor splits a string into an array of lines using the string method split, which cuts up a string at every occurrence of its argument and returns an array of the pieces. The minWidth method finds the maximum line width in this array.

```
function repeat(string, times) {
  var result = "";
  for (var i = 0; i < times; i++)
    result += string;
  return result;
}

function TextCell(text) {
  this.text = text.split("\n");
}
TextCell.prototype.minWidth = function() {
  return this.text.reduce(function(width, line) {
    return Math.max(width, line.length);
  }, 0);
};
TextCell.prototype.minHeight = function() {
  return this.text.length;
};
TextCell.prototype.draw = function(width, height) {
  var result = [];
```

```
  for (var i = 0; i < height; i++) {
    var line = this.text[i] || "";
    result.push(line + repeat(" ", width - line.length));
  }
  return result;
};
```

The code uses a helper function called repeat, which builds a string whose value is the string argument repeated times number of times. The draw method uses it to add "padding" to lines so that they all have the required length.

Let's try everything we've written so far by building up a 5 × 5 checkerboard.

```
var rows = [];
for (var i = 0; i < 5; i++) {
  var row = [];
  for (var j = 0; j < 5; j++) {
    if ((j + i) % 2 == 0)
      row.push(new TextCell("##"));
    else
      row.push(new TextCell("  "));
  }
  rows.push(row);
}
console.log(drawTable(rows));
// → ##    ##    ##
//     ##    ##
//   ##    ##    ##
//     ##    ##
//   ##    ##    ##
```

It works! But since all cells have the same size, the table-layout code doesn't really do anything interesting.

The source data for the table of mountains that we are trying to build is available in the MOUNTAINS variable in the sandbox and also downloadable from the website.

We will want to highlight the top row, which contains the column names, by underlining the cells with a series of dash characters. No problem—we simply write a cell type that handles underlining.

```
function UnderlinedCell(inner) {
  this.inner = inner;
}
UnderlinedCell.prototype.minWidth = function() {
  return this.inner.minWidth();
};
UnderlinedCell.prototype.minHeight = function() {
  return this.inner.minHeight() + 1;
};
UnderlinedCell.prototype.draw = function(width, height) {
  return this.inner.draw(width, height - 1)
    .concat([repeat("-", width)]);
};
```

An underlined cell *contains* another cell. It reports its minimum size as being the same as that of its inner cell (by calling through to that cell's minWidth and minHeight methods) but adds one to the height to account for the space taken up by the underline.

Drawing such a cell is quite simple—we take the content of the inner cell and concatenate a single line full of dashes to it.

Having an underlining mechanism, we can now write a function that builds up a grid of cells from our data set.

```
function dataTable(data) {
  var keys = Object.keys(data[0]);
  var headers = keys.map(function(name) {
    return new UnderlinedCell(new TextCell(name));
  });
  var body = data.map(function(row) {
    return keys.map(function(name) {
      return new TextCell(String(row[name]));
    });
  });
  return [headers].concat(body);
}

console.log(drawTable(dataTable(MOUNTAINS)));
```

```
// → name         height country
//  ------------ ------ -------------
//  Kilimanjaro  5895   Tanzania
//  … etcetera
```

The standard Object.keys function returns an array of property names in an object. The top row of the table must contain underlined cells that give the names of the columns. Below that, the values of all the objects in the data set appear as normal cells—we extract them by mapping over the keys array so that we are sure that the order of the cells is the same in every row.

The resulting table resembles the example shown before, except that it does not right-align the numbers in the height column. We will get to that in a moment.

Getters and setters

When specifying an interface, it is possible to include properties that are not methods. We could have defined minHeight and minWidth to simply hold numbers. But that'd have required us to compute them in the constructor, which adds code there that isn't strictly relevant to *constructing* the object. It would cause problems if, for example, the inner cell of an underlined cell was changed, at which point the size of the underlined cell should also change.

This has led some people to adopt a principle of never including nonmethod properties in interfaces. Rather than directly access a simple value property, they'd use getSomething and setSomething methods to read and write the property. This approach has the downside that you will end up writing—and reading—a lot of additional methods.

Fortunately, JavaScript provides a technique that gets us the best of both worlds. We can specify properties that, from the outside, look like normal properties but secretly have methods associated with them.

```
var pile = {
  elements: ["eggshell", "orange peel", "worm"],
  get height() {
    return this.elements.length;
  },
  set height(value) {
    console.log("Ignoring attempt to set height to", value);
  }
};
```

```
console.log(pile.height);
// → 3
pile.height = 100;
// → Ignoring attempt to set height to 100
```

In an object literal, the get or set notation for properties allows you to specify a function to be run when the property is read or written. You can also add such a property to an existing object, for example a prototype, using the Object.defineProperty function (which we previously used to create nonenumerable properties).

```
Object.defineProperty(TextCell.prototype, "heightProp", {
  get: function() { return this.text.length; }
});

var cell = new TextCell("no\nway");
console.log(cell.heightProp);
// → 2
cell.heightProp = 100;
console.log(cell.heightProp);
// → 2
```

You can use a similar set property, in the object passed to defineProperty, to specify a setter method. When a getter but no setter is defined, writing to the property is simply ignored.

Inheritance

We are not quite done yet with our table layout exercise. It helps readability to right-align columns of numbers. We should create another cell type that is like TextCell, but rather than padding the lines on the right side, it pads them on the left side so that they align to the right.

We could simply write a whole new constructor with all three methods in its prototype. But prototypes may themselves have prototypes, and this allows us to do something clever.

```
function RTextCell(text) {
  TextCell.call(this, text);
}
RTextCell.prototype = Object.create(TextCell.prototype);
RTextCell.prototype.draw = function(width, height) {
```

```
  var result = [];
  for (var i = 0; i < height; i++) {
    var line = this.text[i] || "";
    result.push(repeat(" ", width - line.length) + line);
  }
  return result;
};
```

We reuse the constructor and the minHeight and minWidth methods from the regular TextCell. An RTextCell is now basically equivalent to a TextCell, except that its draw method contains a different function.

This pattern is called *inheritance*. It allows us to build slightly different data types from existing data types with relatively little work. Typically, the new constructor will call the old constructor (using the call method in order to be able to give it the new object as its this value). Once this constructor has been called, we can assume that all the fields that the old object type is supposed to contain have been added. We arrange for the constructor's prototype to derive from the old prototype so that instances of this type will also have access to the properties in that prototype. Finally, we can override some of these properties by adding them to our new prototype.

Now, if we slightly adjust the dataTable function to use RTextCells for cells whose value is a number, we get the table we were aiming for.

```
function dataTable(data) {
  var keys = Object.keys(data[0]);
  var headers = keys.map(function(name) {
    return new UnderlinedCell(new TextCell(name));
  });
  var body = data.map(function(row) {
    return keys.map(function(name) {
      var value = row[name];
      // This was changed:
      if (typeof value == "number")
        return new RTextCell(String(value));
      else
        return new TextCell(String(value));
    });
  });
```

```
  return [headers].concat(body);
}

console.log(drawTable(dataTable(MOUNTAINS)));
// → … beautifully aligned table
```

Inheritance is a fundamental part of the object-oriented tradition, alongside encapsulation and polymorphism. But while the latter two are now generally regarded as wonderful ideas, inheritance is somewhat controversial.

The main reason for this is that it is often confused with polymorphism, sold as a more powerful tool than it really is, and subsequently overused in all kinds of ugly ways. Whereas encapsulation and polymorphism can be used to *separate* pieces of code from each other, reducing the tangledness of the overall program, inheritance fundamentally ties types together, creating *more* tangle.

You can have polymorphism without inheritance, as we saw. I am not going to tell you to avoid inheritance entirely—I use it regularly in my own programs. But you should see it as a slightly dodgy trick that can help you define new types with little code, not as a grand principle of code organization. A preferable way to extend types is through composition, such as how UnderlinedCell builds on another cell object by simply storing it in a property and forwarding method calls to it in its own methods.

The instanceof operator

It is occasionally useful to know whether an object was derived from a specific constructor. For this, JavaScript provides a binary operator called instanceof.

```
console.log(new RTextCell("A") instanceof RTextCell);
// → true
console.log(new RTextCell("A") instanceof TextCell);
// → true
console.log(new TextCell("A") instanceof RTextCell);
// → false
console.log([1] instanceof Array);
// → true
```

The operator will see through inherited types. An RTextCell is an instance of TextCell because RTextCell.prototype derives from TextCell.prototype. The operator can be applied to standard constructors like Array. Almost every object is an instance of Object.

Summary

So objects are more complicated than I initially portrayed them. They have prototypes, which are other objects, and will act as if they have properties they don't have as long as the prototype has that property. Simple objects have Object.prototype as their prototype.

Constructors, which are functions whose names usually start with a capital letter, can be used with the new operator to create new objects. The new object's prototype will be the object found in the prototype property of the constructor function. You can make good use of this by putting the properties that all values of a given type share into their prototype. The instanceof operator can, given an object and a constructor, tell you whether that object is an instance of that constructor.

One useful thing to do with objects is to specify an interface for them and tell everybody that they are supposed to talk to your object only through that interface. The rest of the details that make up your object are now *encapsulated*, hidden behind the interface.

Once you are talking in terms of interfaces, who says that only one kind of object may implement this interface? Having different objects expose the same interface and then writing code that works on any object with the interface is called *polymorphism*. It is very useful.

When implementing multiple types that differ in only some details, it can be helpful to simply make the prototype of your new type derive from the prototype of your old type and have your new constructor call the old one. This gives you an object type similar to the old type but for which you can add and override properties as you see fit.

Exercises

A vector type

Write a constructor Vector that represents a vector in two-dimensional space. It takes x and y parameters (numbers), which it should save to properties of the same name.

Give the Vector prototype two methods, plus and minus, that take another vector as a parameter and return a new vector that has the sum or difference of the two vectors' (the one in this and the parameter) *x* and *y* values.

Add a getter property length to the prototype that computes the length of the vector—that is, the distance of the point (*x*, *y*) from the origin (0, 0).

```
// Your code here.

console.log(new Vector(1, 2).plus(new Vector(2, 3)));
// → Vector{x: 3, y: 5}
console.log(new Vector(1, 2).minus(new Vector(2, 3)));
// → Vector{x: -1, y: -1}
console.log(new Vector(3, 4).length);
// → 5
```

Another cell

Implement a cell type named StretchCell(inner, width, height) that conforms to the table cell interface described earlier in the chapter. It should wrap another cell (like UnderlinedCell does) and ensure that the resulting cell has at least the given width and height, even if the inner cell would naturally be smaller.

```
// Your code here.

var sc = new StretchCell(new TextCell("abc"), 1, 2);
console.log(sc.minWidth());
// → 3
console.log(sc.minHeight());
// → 2
console.log(sc.draw(3, 2));
// → ["abc", "   "]
```

Sequence interface

Design an *interface* that abstracts iteration over a collection of values. An object that provides this interface represents a sequence, and the interface must somehow make it possible for code that uses such an object to iterate over the sequence, looking at the element values it is made up of and having some way to find out when the end of the sequence is reached.

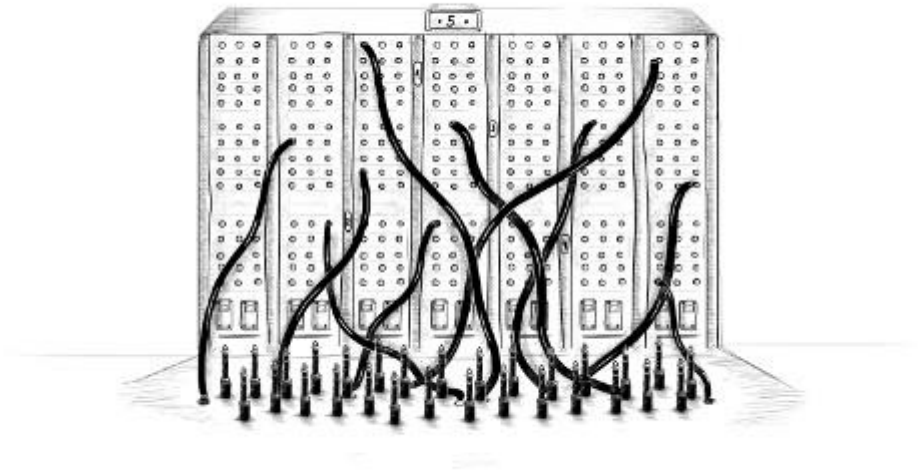
When you have specified your interface, try to write a function logFive that takes a sequence object and calls console.log on its first five elements—or fewer, if the sequence has fewer than five elements.

Then implement an object type ArraySeq that wraps an array and allows iteration over the array using the interface you designed. Implement another object type RangeSeq that iterates over a range of integers (taking from and to arguments to its constructor) instead.

JavaScript and the Browser

The dream behind the Web is of a common information space in which we communicate by sharing information. Its universality is essential: the fact that a hypertext link can point to anything, be it personal, local or global, be it draft or highly polished.

Tim Berners-Lee, *The World Wide Web: A very short personal history*



The next chapters of this book will talk about web browsers. Without web browsers, there would be no JavaScript. Or even if there were, no one would ever have paid any attention to it.

Web technology has been decentralized from the start, not just technically but also in the way it evolved. Various browser vendors have added new functionality in ad hoc and sometimes poorly thought-out ways, which then, sometimes, ended up being adopted by others—and finally set down as in standards.

This is both a blessing and a curse. On the one hand, it is empowering to not have a central party control a system but have it be improved by various parties working in loose collaboration (or occasionally open hostility). On the other hand, the haphazard way in which the Web was developed means that the resulting system is not exactly a shining example of internal consistency. Some parts of it are downright confusing and poorly conceived.

Networks and the Internet

Computer networks have been around since the 1950s. If you put cables between two or more computers and allow them to send data back and forth through these cables, you can do all kinds of wonderful things.

And if connecting two machines in the same building allows us to do wonderful things, connecting machines all over the planet should be even better. The technology to start implementing this vision was developed in the 1980s, and the resulting network is called the *Internet*. It has lived up to its promise.

A computer can use this network to shoot bits at another computer. For any effective communication to arise out of this bit-shooting, the computers on both ends must know what the bits are supposed to represent. The meaning of any given sequence of bits depends entirely on the kind of thing that it is trying to express and on the encoding mechanism used.

A *network protocol* describes a style of communication over a network. There are protocols for sending email, for fetching email, for sharing files, and even for controlling computers that happen to be infected by malicious software.

For example, the *Hypertext Transfer Protocol* (HTTP) is a protocol for retrieving named resources (chunks of information, such as web pages or pictures). It specifies that the side making the request should start with a line like this, naming the resource and the version of the protocol that it is trying to use:

```
GET /index.html HTTP/1.1
```

There are a lot more rules about the way the requester can include more information in the request and the way the other side, which returns the resource, packages up its content. We'll look at HTTP in a little more detail in Chapter 18.

Most protocols are built on top of other protocols. HTTP treats the network as a streamlike device into which you can put bits and have them arrive at the correct destination in the correct order. As we saw in Chapter 11, ensuring those things is already a rather difficult problem.

The *Transmission Control Protocol* (TCP) is a protocol that addresses this problem. All Internet-connected devices "speak" it, and most communication on the Internet is built on top of it.

A TCP connection works as follows: one computer must be waiting, or *listening*, for other computers to start talking to it. To be able to listen for different kinds of communication at the same time on a single machine, each listener has a number (called a *port*) associated with it. Most protocols specify which port should be used by default. For example, when we want to send an email using the SMTP protocol, the machine through which we send it is expected to be listening on port 25.

Another computer can then establish a connection by connecting to the target machine using the correct port number. If the target machine can be reached and is listening on that port, the connection is successfully created. The listening computer is called the *server*, and the connecting computer is called the *client*.

Such a connection acts as a two-way pipe through which bits can flow—the machines on both ends can put data into it. Once the bits are successfully transmitted, they can be read out again by the machine on the other side. This is a convenient model. You could say that TCP provides an abstraction of the network.

The Web

The *World Wide Web* (not to be confused with the Internet as a whole) is a set of protocols and formats that allow us to visit web pages in a browser. The "Web" part in the name refers to the fact that such pages can easily link to each other, thus connecting into a huge mesh that users can move through.

To become part of the Web, all you need to do is connect a machine to the Internet and have it listen on port 80 with the HTTP protocol so that other computers can ask it for documents.

Each document on the Web is named by a *Uniform Resource Locator* (URL), which looks something like this:

```
  http://eloquentjavascript.net/13_browser.html
|    |                  |           |
protocol       server            path
```

The first part tells us that this URL uses the HTTP protocol (as opposed to, for example, encrypted HTTP, which would be *https://*). Then comes the part that identifies which server we are requesting the document from. Last is a path string that identifies the specific document (or *resource*) we are interested in.

Machines connected to the Internet get an *IP address*, which is a number that can be used to send messages to that machine, and looks something like 149.210.142.219 or 2001:4860:4860::8888. But lists of more or less random numbers are hard to remember and awkward to type, so you can instead register a *domain name* for a specific address or set of addresses. I registered *eloquentjavascript.net* to point at the IP address of a machine I control and can thus use that domain name to serve web pages.

If you type this URL into your browser's address bar, the browser will try to retrieve and display the document at that URL. First, your browser has to find out what address *eloquentjavascript.net* refers to. Then, using the HTTP protocol, it will make a connection to the server at that address and ask for the resource */13_browser.html*. If all goes well, the server sends back a document, which your browser then displays on your screen.

HTML

HTML, which stands for *Hypertext Markup Language*, is the document format used for web pages. An HTML document contains text, as well as *tags* that give structure to the text, describing things such as links, paragraphs, and headings.

A short HTML document might look like this:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentjavascript.net">here</a>.</p>
  </body>
</html>
```

The tags, wrapped in angle brackets (< and >, the symbols for *less than* and *greater than*), provide information about the structure of the document. The other text is just plain text.

The document starts with <!doctype html>, which tells the browser to interpret the page as *modern* HTML, as opposed to various dialects that were in use in the past.

HTML documents have a head and a body. The head contains information *about* the document, and the body contains the document itself. In this case, the head declares that the title of this document is “My home page” and that it uses the UTF-8 encoding, which is a way to encode Unicode text as binary data. The document’s body contains a heading (<h1>, meaning “heading 1”—<h2> to <h6> produce subheadings) and two paragraphs ( ).

Tags come in several forms. An element, such as the body, a paragraph, or a link, is started by an *opening tag* like and ended by a *closing tag* like . Some opening tags, such as the one for the link (<a>), contain extra information in the form of name="value" pairs. These are called *attributes*. In this case, the

destination of the link is indicated with href="http://eloquentjavascript.net", where href stands for “hypertext reference”.

Some kinds of tags do not enclose anything and thus do not need to be closed. The metadata tag <meta charset="utf-8"> is an example of this.

To be able to include angle brackets in the text of a document, even though they have a special meaning in HTML, yet another form of special notation has to be introduced. A plain opening angle bracket is written as < (“less than”), and a closing bracket is written as > (“greater than”). In HTML, an ampersand (&) character followed by a name or character code and a semicolon (;) is called an *entity* and will be replaced by the character it encodes.

This is analogous to the way backslashes are used in JavaScript strings. Since this mechanism gives ampersand characters a special meaning, too, they need to be escaped as &. Inside attribute values, which are wrapped in double quotes, " can be used to insert an actual quote character.

HTML is parsed in a remarkably error-tolerant way. When tags that should be there are missing, the browser reconstructs them. The way in which this is done has been standardized, and you can rely on all modern browsers to do it in the same way.

The following document will be treated just like the one shown previously:

```
<!doctype html>

<meta charset=utf-8>
<title>My home page</title>

<h1>My home page</h1>
<p>Hello, I am Marijn and this is my home page.
<p>I also wrote a book! Read it
  <a href=http://eloquentjavascript.net>here</a>.
```

The <html>, <head>, and <body> tags are gone completely. The browser knows that <meta> and <title> belong in the head and that <h1> means the body has started. Furthermore, I am no longer explicitly closing the paragraphs since opening a new paragraph or ending the document will close them implicitly. The quotes around the attribute values are also gone.

This book will usually omit the <html>, <head>, and <body> tags from examples to keep them short and free of clutter. But I *will* close tags and include quotes around attributes.

I will also usually omit the doctype and charset declaration. This is not to be taken as an encouragement to drop these from HTML documents. Browsers will often do ridiculous things when you forget them. You should consider the doctype and the charset metadata to be implicitly present in examples, even when they are not actually shown in the text.

HTML and JavaScript

In the context of this book, the most important HTML tag is <script>. This tag allows us to include a piece of JavaScript in a document.

```
<h1>Testing alert</h1>
<script>alert("hello!");</script>
```

Such a script will run as soon as its <script> tag is encountered while the browser reads the HTML. This page will pop up a dialog when opened—the alert function resembles prompt, in that it pops up a little window, but only shows a message without asking for input.

Including large programs directly in HTML documents is often impractical. The <script> tag can be given an src attribute to fetch a script file (a text file containing a JavaScript program) from a URL.

```
<h1>Testing alert</h1>
<script src="code/hello.js"></script>
```

The *code/hello.js* file included here contains the same program—alert("hello!"). When an HTML page references other URLs as part of itself—for example, an image file or a script—web browsers will retrieve them immediately and include them in the page.

A script tag must always be closed with </script>, even if it refers to a script file and doesn't contain any code. If you forget this, the rest of the page will be interpreted as part of the script.

You can load ES modules (see Chapter 10) in the browser by giving your script tag a type="module" attribute. Such modules can depend on other modules by using URLs relative to themselves as module names in import declarations.

Some attributes can also contain a JavaScript program. The <button> tag shown next (which shows up as a button) has an onclick attribute. The attribute's value will be run whenever the button is clicked.

```
<button onclick="alert('Boom!');">DO NOT PRESS</button>
```

Note that I had to use single quotes for the string in the onclick attribute because double quotes are already used to quote the whole attribute. I could also have used ".

In the sandbox

Running programs downloaded from the Internet is potentially dangerous. You do not know much about the people behind most sites you visit, and they do not necessarily mean well. Running programs by people who do not mean well is how you get your computer infected by viruses, your data stolen, and your accounts hacked.

Yet the attraction of the Web is that you can browse it without necessarily trusting all the pages you visit. This is why browsers severely limit the things a JavaScript program may do: it can't look at the files on your computer or modify anything not related to the web page it was embedded in.

Isolating a programming environment in this way is called *sandboxing*, the idea being that the program is harmlessly playing in a sandbox. But you should imagine this particular kind of sandbox as having a cage of thick steel bars over it so that the programs playing in it can't actually get out.

The hard part of sandboxing is allowing the programs enough room to be useful yet at the same time restricting them from doing anything dangerous. Lots of useful functionality, such as communicating with other servers or reading the content of the copy-paste clipboard, can also be used to do problematic, privacy-invading things.

Every now and then, someone comes up with a new way to circumvent the limitations of a browser and do something harmful, ranging from leaking minor private information to taking over the whole machine that the browser runs on. The browser developers respond by fixing the hole, and all is well again—until the next problem is discovered, and hopefully publicized, rather than secretly exploited by some government agency or mafia.

Compatibility and the browser wars

In the early stages of the Web, a browser called Mosaic dominated the market. After a few years, the balance shifted to Netscape, which was then, in turn, largely supplanted by Microsoft's Internet Explorer. At any point where a single browser was dominant, that browser's vendor would feel entitled to unilaterally invent new features for the Web. Since most users used the most popular browser, websites would simply start using those features—never mind the other browsers.

This was the dark age of compatibility, often called the *browser wars*. Web developers were left with not one unified Web but two or three incompatible platforms. To make things worse, the browsers in use around

2003 were all full of bugs, and of course the bugs were different for each browser. Life was hard for people writing web pages.

Mozilla Firefox, a not-for-profit offshoot of Netscape, challenged Internet Explorer's position in the late 2000s. Because Microsoft was not particularly interested in staying competitive at the time, Firefox took a lot of market share away from it. Around the same time, Google introduced its Chrome browser, and Apple's Safari browser gained popularity, leading to a situation where there were four major players, rather than one.

The new players had a more serious attitude toward standards and better engineering practices, giving us less incompatibility and fewer bugs. Microsoft, seeing its market share crumble, came around and adopted these attitudes in its Edge browser, which replaces Internet Explorer. If you are starting to learn web development today, consider yourself lucky. The latest versions of the major browsers behave quite uniformly and have relatively few bugs.

The Document Object Model

Too bad! Same old story! Once you've finished building your house you notice you've accidentally learned something that you really should have known—before you started.

Friedrich Nietzsche, *Beyond Good and Evil*

When you open a web page in your browser, the browser retrieves the page's HTML text and parses it, much like the way our parser from Chapter 12 parsed programs. The browser builds up a model of the document's structure and uses this model to draw the page on the screen.

This representation of the document is one of the toys that a JavaScript program has available in its sandbox. It is a data structure that you can read or modify. It acts as a *live* data structure: when it's modified, the page on the screen is updated to reflect the changes.

Document structure

You can imagine an HTML document as a nested set of boxes. Tags such as <body> and </body> enclose other tags, which in turn contain other tags or text. Here's the example document from the previous chapter:

```
<!doctype html>
<html>
  <head>
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentjavascript.net">here</a>.</p>
  </body>
</html>
```

This page has the following structure:

The data structure the browser uses to represent the document follows this shape. For each box, there is an object, which we can interact with to find out things such as what HTML tag it represents and which boxes and text it contains. This representation is called the *Document Object Model*, or DOM for short.

The global binding document gives us access to these objects. Its documentElement property refers to the object representing the <html> tag. Since every HTML document has a head and a body, it also has head and body properties, pointing at those elements.

Trees

Think back to the syntax trees from Chapter 12 for a moment. Their structures are strikingly similar to the structure of a browser's document. Each *node* may refer to other nodes, *children*, which in turn may have

their own children. This shape is typical of nested structures where elements can contain subelements that are similar to themselves.

We call a data structure a *tree* when it has a branching structure, has no cycles (a node may not contain itself, directly or indirectly), and has a single, well-defined *root*. In the case of the DOM, document.documentElement serves as the root.

Trees come up a lot in computer science. In addition to representing recursive structures such as HTML documents or programs, they are often used to maintain sorted sets of data because elements can usually be found or inserted more efficiently in a tree than in a flat array.

A typical tree has different kinds of nodes. The syntax tree for the Egg language had identifiers, values, and application nodes. Application nodes may have children, whereas identifiers and values are *leaves*, or nodes without children.

The same goes for the DOM. Nodes for *elements*, which represent HTML tags, determine the structure of the document. These can have child nodes. An example of such a node is document.body. Some of these children can be leaf nodes, such as pieces of text or comment nodes.

Each DOM node object has a nodeType property, which contains a code (number) that identifies the type of node. Elements have code 1, which is also defined as the constant property Node.ELEMENT_NODE. Text nodes, representing a section of text in the document, get code 3 (Node.TEXT_NODE). Comments have code 8 (Node.COMMENT_NODE).

Another way to visualize our document tree is as follows:

The leaves are text nodes, and the arrows indicate parent-child relationships between nodes.

The standard

Using cryptic numeric codes to represent node types is not a very JavaScript-like thing to do. Later in this chapter, we'll see that other parts of the DOM interface also feel cumbersome and alien. The reason for this is that the DOM wasn't designed for just JavaScript. Rather, it tries to be a language-neutral interface that can be used in other systems as well—not just for HTML but also for XML, which is a generic data format with an HTML-like syntax.

This is unfortunate. Standards are often useful. But in this case, the advantage (cross-language consistency) isn't all that compelling. Having an interface that is properly integrated with the language you are using will save you more time than having a familiar interface across languages.

As an example of this poor integration, consider the childNodes property that element nodes in the DOM have. This property holds an array-like object, with a length property and properties labeled by numbers to access the child nodes. But it is an instance of the NodeList type, not a real array, so it does not have methods such as slice and map.

Then there are issues that are simply poor design. For example, there is no way to create a new node and immediately add children or attributes to it. Instead, you have to first create it and then add the children and attributes one by one, using side effects. Code that interacts heavily with the DOM tends to get long, repetitive, and ugly.

But these flaws aren't fatal. Since JavaScript allows us to create our own abstractions, it is possible to design improved ways to express the operations you are performing. Many libraries intended for browser programming come with such tools.

Moving through the tree

DOM nodes contain a wealth of links to other nearby nodes. The following diagram illustrates these:

Although the diagram shows only one link of each type, every node has a parentNode property that points to the node it is part of, if any. Likewise, every element node (node type 1) has a childNodes property that points to an array-like object holding its children.

In theory, you could move anywhere in the tree using just these parent and child links. But JavaScript also gives you access to a number of additional convenience links. The firstChild and lastChild properties point to the first and last child elements or have the value null for nodes without children. Similarly, previousSibling and nextSibling point to adjacent nodes, which are nodes with the same parent that appear immediately before or after the node itself. For a first child, previousSibling will be null, and for a last child, nextSibling will be null.

There's also the children property, which is like childNodes but contains only element (type 1) children, not other types of child nodes. This can be useful when you aren't interested in text nodes.

When dealing with a nested data structure like this one, recursive functions are often useful. The following function scans a document for text nodes containing a given string and returns true when it has found one:

```
function talksAbout(node, string) {
  if (node.nodeType == Node.ELEMENT_NODE) {
    for (let child of node.childNodes) {
      if (talksAbout(child, string)) {
        return true;
      }
    }
    return false;
  } else if (node.nodeType == Node.TEXT_NODE) {
    return node.nodeValue.indexOf(string) > -1;
  }
}

console.log(talksAbout(document.body, "book"));
// → true
```

The nodeValue property of a text node holds the string of text that it represents.

Finding elements

Navigating these links among parents, children, and siblings is often useful. But if we want to find a specific node in the document, reaching it by starting at document.body and following a fixed path of properties is a bad idea. Doing so bakes assumptions into our program about the precise structure of the document—a structure you might want to change later. Another complicating factor is that text nodes are created even for the whitespace between nodes. The example document's <body> tag does not have just three children (<h1> and two <p> elements) but actually has seven: those three, plus the spaces before, after, and between them.

So if we want to get the href attribute of the link in that document, we don't want to say something like "Get the second child of the sixth child of the document body". It'd be better if we could say "Get the first link in the document". And we can.

```
let link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

All element nodes have a getElementsByTagName method, which collects all elements with the given tag name that are descendants (direct or indirect children) of that node and returns them as an array-like object.

To find a specific *single* node, you can give it an id attribute and use document.getElementById instead.

```
<p>My ostrich Gertrude:</p>
<p><img id="gertrude" src="img/ostrich.png"></p>

<script>
  let ostrich = document.getElementById("gertrude");
  console.log(ostrich.src);
</script>
```

A third, similar method is getElementsByClassName, which, like getElementsByTagName, searches through the contents of an element node and retrieves all elements that have the given string in their class attribute.

Changing the document

Almost everything about the DOM data structure can be changed. The shape of the document tree can be modified by changing parent-child relationships. Nodes have a remove method to remove them from their current parent node. To add a child node to an element node, we can use appendChild, which puts it at the end of the list of children, or insertBefore, which inserts the node given as the first argument before the node given as the second argument.

```
<p>One</p>
<p>Two</p>
<p>Three</p>

<script>
  let paragraphs = document.body.getElementsByTagName("p");
  document.body.insertBefore(paragraphs[2], paragraphs[0]);
</script>
```

A node can exist in the document in only one place. Thus, inserting paragraph *Three* in front of paragraph *One* will first remove it from the end of the document and then insert it at the front, resulting in *Three*/*One*/*Two*. All operations that insert a node somewhere will, as a side effect, cause it to be removed from its current position (if it has one).

The replaceChild method is used to replace a child node with another one. It takes as arguments two nodes: a new node and the node to be replaced. The replaced node must be a child of the element the method is called on. Note that both replaceChild and insertBefore expect the *new* node as their first argument.

Creating nodes

Say we want to write a script that replaces all images (<img> tags) in the document with the text held in their alt attributes, which specifies an alternative textual representation of the image.

This involves not only removing the images but adding a new text node to replace them. Text nodes are created with the document.createTextNode method.

```
<p>The <img src="img/cat.png" alt="Cat"> in the
  <img src="img/hat.png" alt="Hat">.</p>

<p><button onclick="replaceImages()">Replace</button></p>

<script>
  function replaceImages() {
    let images = document.body.getElementsByTagName("img");
    for (let i = images.length - 1; i >= 0; i--) {
      let image = images[i];
      if (image.alt) {
        let text = document.createTextNode(image.alt);
        image.parentNode.replaceChild(text, image);
      }
    }
  }
</script>
```

Given a string, createTextNode gives us a text node that we can insert into the document to make it show up on the screen.

The loop that goes over the images starts at the end of the list. This is necessary because the node list returned by a method like getElementsByTagName (or a property like childNodes) is *live*. That is, it is updated as the document changes. If we started from the front, removing the first image would cause the list to lose its first element so that the second time the loop repeats, where i is 1, it would stop because the length of the collection is now also 1.

If you want a *solid* collection of nodes, as opposed to a live one, you can convert the collection to a real array by calling Array.from.

```
let arrayish = {0: "one", 1: "two", length: 2};
let array = Array.from(arrayish);
console.log(array.map(s => s.toUpperCase()));
// → ["ONE", "TWO"]
```

To create element nodes, you can use the document.createElement method. This method takes a tag name and returns a new empty node of the given type.

The following example defines a utility elt, which creates an element node and treats the rest of its arguments as children to that node. This function is then used to add an attribution to a quote.

```
<blockquote id="quote">
  No book can ever be finished. While working on it we learn
  just enough to find it immature the moment we turn away
  from it.
</blockquote>

<script>
  function elt(type, ...children) {
    let node = document.createElement(type);
    for (let child of children) {
      if (typeof child != "string") node.appendChild(child);
      else node.appendChild(document.createTextNode(child));
    }
    return node;
  }

  document.getElementById("quote").appendChild(
    elt("footer", "—",
        elt("strong", "Karl Popper"),
        ", preface to the second edition of ",
        elt("em", "The Open Society and Its Enemies"),
        ", 1950"));
</script>
```

Attributes

Some element attributes, such as href for links, can be accessed through a property of the same name on the element's DOM object. This is the case for most commonly used standard attributes.

But HTML allows you to set any attribute you want on nodes. This can be useful because it allows you to store extra information in a document. If you make up your own attribute names, though, such attributes will not be present as properties on the element's node. Instead, you have to use the getAttribute and setAttribute methods to work with them.

```
<p data-classified="secret">The launch code is 00000000.</p>
<p data-classified="unclassified">I have two feet.</p>

<script>
  let paras = document.body.getElementsByTagName("p");
  for (let para of Array.from(paras)) {
    if (para.getAttribute("data-classified") == "secret") {
      para.remove();
    }
  }
</script>
```

It is recommended to prefix the names of such made-up attributes with data- to ensure they do not conflict with any other attributes.

There is a commonly used attribute, class, which is a keyword in the JavaScript language. For historical reasons—some old JavaScript implementations could not handle property names that matched keywords—the property used to access this attribute is called className. You can also access it under its real name, "class", by using the getAttribute and setAttribute methods.

Layout

You may have noticed that different types of elements are laid out differently. Some, such as paragraphs (<p>) or headings (<h1>), take up the whole width of the document and are rendered on separate lines. These are called *block* elements. Others, such as links (<a>) or the element, are rendered on the same line with their surrounding text. Such elements are called *inline* elements.

For any given document, browsers are able to compute a layout, which gives each element a size and position based on its type and content. This layout is then used to actually draw the document.

The size and position of an element can be accessed from JavaScript. The offsetWidth and offsetHeight properties give you the space the element takes up in *pixels*. A pixel is the basic unit of measurement in the browser. It traditionally corresponds to the smallest dot that the screen can draw, but on modern displays, which can draw *very* small dots, that may no longer be the case, and a browser pixel may span multiple display dots.

Similarly, clientWidth and clientHeight give you the size of the space *inside* the element, ignoring border width.

```
<p style="border: 3px solid red">
  I'm boxed in
</p>

<script>
  let para = document.body.getElementsByTagName("p")[0];
  console.log("clientHeight:", para.clientHeight);
  console.log("offsetHeight:", para.offsetHeight);
</script>
```

The most effective way to find the precise position of an element on the screen is the getBoundingClientRect method. It returns an object with top, bottom, left, and right properties, indicating the pixel positions of the sides of the element relative to the top left of the screen. If you want them relative to the whole document, you must add the current scroll position, which you can find in the pageXOffset and pageYOffset bindings.

Laying out a document can be quite a lot of work. In the interest of speed, browser engines do not immediately re-layout a document every time you change it but wait as long as they can. When a JavaScript program that changed the document finishes running, the browser will have to compute a new layout to draw the changed document to the screen. When a program *asks* for the position or size of something by reading properties such as offsetHeight or calling getBoundingClientRect, providing correct information also requires computing a layout.

A program that repeatedly alternates between reading DOM layout information and changing the DOM forces a lot of layout computations to happen and will consequently run very slowly. The following code is an example of this. It contains two different programs that build up a line of *X* characters 2,000 pixels wide and measures the time each one takes.

```
<p><span id="one"></span></p>
```

```
<p><span id="two"></span></p>

<script>
  function time(name, action) {
    let start = Date.now(); // Current time in milliseconds
    action();
    console.log(name, "took", Date.now() - start, "ms");
  }

  time("naive", () => {
    let target = document.getElementById("one");
    while (target.offsetWidth < 2000) {
      target.appendChild(document.createTextNode("X"));
    }
  });
  // → naive took 32 ms

  time("clever", function() {
    let target = document.getElementById("two");
    target.appendChild(document.createTextNode("XXXXX"));
    let total = Math.ceil(2000 / (target.offsetWidth / 5));
    target.firstChild.nodeValue = "X".repeat(total);
  });
  // → clever took 1 ms
</script>
```

Styling

We have seen that different HTML elements are drawn differently. Some are displayed as blocks, others inline. Some add styling— makes its content bold, and <a> makes it blue and underlines it.

The way an <img> tag shows an image or an <a> tag causes a link to be followed when it is clicked is strongly tied to the element type. But we can change the styling associated with an element, such as the text color or underline. Here is an example that uses the style property:

```
<p><a href=".">Normal link</a></p>
<p><a href="." style="color: green">Green link</a></p>
```

A style attribute may contain one or more *declarations*, which are a property (such as color) followed by a colon and a value (such as green). When there is more than one declaration, they must be separated by semicolons, as in "color: red; border: none".

A lot of aspects of the document can be influenced by styling. For example, the display property controls whether an element is displayed as a block or an inline element.

```
This text is displayed <strong>inline</strong>,
<strong style="display: block">as a block</strong>, and
<strong style="display: none">not at all</strong>.
```

The block tag will end up on its own line since block elements are not displayed inline with the text around them. The last tag is not displayed at all—display: none prevents an element from showing up on the screen. This is a way to hide elements. It is often preferable to removing them from the document entirely because it makes it easy to reveal them again later.

JavaScript code can directly manipulate the style of an element through the element's style property. This property holds an object that has properties for all possible style properties. The values of these properties are strings, which we can write to in order to change a particular aspect of the element's style.

```
<p id="para" style="color: purple">
  Nice text
</p>

<script>
  let para = document.getElementById("para");
  console.log(para.style.color);
  para.style.color = "magenta";
</script>
```

Some style property names contain hyphens, such as font-family. Because such property names are awkward to work with in JavaScript (you'd have to say style["font-family"]), the property names in the style object for such properties have their hyphens removed and the letters after them capitalized (style.fontFamily).

Cascading styles

The styling system for HTML is called CSS, for *Cascading Style Sheets*. A *style sheet* is a set of rules for how to style elements in a document. It can be given inside a <style> tag.

```
<style>
  strong {
    font-style: italic;
    color: gray;
  }
</style>
<p>Now <strong>strong text</strong> is italic and gray.</p>
```

The *cascading* in the name refers to the fact that multiple such rules are combined to produce the final style for an element. In the example, the default styling for tags, which gives them font-weight: bold, is overlaid by the rule in the <style> tag, which adds font-style and color.

When multiple rules define a value for the same property, the most recently read rule gets a higher precedence and wins. So if the rule in the <style> tag included font-weight: normal, contradicting the default font-weight rule, the text would be normal, *not* bold. Styles in a style attribute applied directly to the node have the highest precedence and always win.

It is possible to target things other than tag names in CSS rules. A rule for .abc applies to all elements with "abc" in their class attribute. A rule for #xyz applies to the element with an id attribute of "xyz" (which should be unique within the document).

```
.subtle {
  color: gray;
  font-size: 80%;
}
#header {
  background: blue;
  color: white;
}
/* p elements with id main and with classes a and b */
p#main.a.b {
  margin-bottom: 20px;
}
```

The precedence rule favoring the most recently defined rule applies only when the rules have the same *specificity*. A rule's specificity is a measure of how precisely it describes matching elements, determined by the number and kind (tag, class, or ID) of element aspects it requires. For example, a rule that targets p.a is more specific than rules that target p or just .a and would thus take precedence over them.

The notation p > a {…} applies the given styles to all <a> tags that are direct children of <p> tags. Similarly, p a {…} applies to all <a> tags inside <p> tags, whether they are direct or indirect children.

Query selectors

We won't be using style sheets all that much in this book. Understanding them is helpful when programming in the browser, but they are complicated enough to warrant a separate book.

The main reason I introduced *selector* syntax—the notation used in style sheets to determine which elements a set of styles apply to—is that we can use this same mini-language as an effective way to find DOM elements.

The querySelectorAll method, which is defined both on the document object and on element nodes, takes a selector string and returns a NodeList containing all the elements that it matches.

```
<p>And if you go chasing
  <span class="animal">rabbits</span></p>
<p>And you know you're going to fall</p>
<p>Tell 'em a <span class="character">hookah smoking
  <span class="animal">caterpillar</span></span></p>
<p>Has given you the call</p>

<script>
  function count(selector) {
    return document.querySelectorAll(selector).length;
  }
  console.log(count("p"));           // All <p> elements
  // → 4
  console.log(count(".animal"));     // Class animal
  // → 2
  console.log(count("p .animal"));   // Animal inside of <p>
  // → 2
  console.log(count("p > .animal")); // Direct child of <p>
  // → 1
</script>
```

Unlike methods such as getElementsByTagName, the object returned by querySelectorAll is *not* live. It won't change when you change the document. It is still not a real array, though, so you still need to call Array.from if you want to treat it like one.

The querySelector method (without the All part) works in a similar way. This one is useful if you want a specific, single element. It will return only the first matching element or null when no element matches.

Positioning and animating

The position style property influences layout in a powerful way. By default it has a value of static, meaning the element sits in its normal place in the document. When it is set to relative, the element still takes up space in the document, but now the top and left style properties can be used to move it relative to that normal place. When position is set to absolute, the element is removed from the normal document flow—that is, it no longer takes up space and may overlap with other elements. Also, its top and left properties can be used to absolutely position it relative to the top-left corner of the nearest enclosing element whose position property isn't static, or relative to the document if no such enclosing element exists.

We can use this to create an animation. The following document displays a picture of a cat that moves around in an ellipse:

```
<p style="text-align: center">
  <img src="img/cat.png" style="position: relative">
</p>
<script>
  let cat = document.querySelector("img");
  let angle = Math.PI / 2;
  function animate(time, lastTime) {
    if (lastTime != null) {
      angle += (time - lastTime) * 0.001;
    }
    cat.style.top = (Math.sin(angle) * 20) + "px";
    cat.style.left = (Math.cos(angle) * 200) + "px";
    requestAnimationFrame(newTime => animate(newTime, time));
  }
  requestAnimationFrame(animate);
</script>
```

Our picture is centered on the page and given a position of relative. We'll repeatedly update that picture's top and left styles to move it.

The script uses requestAnimationFrame to schedule the animate function to run whenever the browser is ready to repaint the screen. The animate function itself again calls requestAnimationFrame to schedule the next update. When the browser window (or tab) is active, this will cause updates to happen at a rate of about 60 per second, which tends to produce a good-looking animation.

If we just updated the DOM in a loop, the page would freeze, and nothing would show up on the screen. Browsers do not update their display while a JavaScript program is running, nor do they allow any interaction with the page. This is why we need requestAnimationFrame—it lets the browser know that we are done for now, and it can go ahead and do the things that browsers do, such as updating the screen and responding to user actions.

The animation function is passed the current time as an argument. To ensure that the motion of the cat per millisecond is stable, it bases the speed at which the angle changes on the difference between the current time and the last time the function ran. If it just moved the angle by a fixed amount per step, the motion would stutter if, for example, another heavy task running on the same computer were to prevent the function from running for a fraction of a second.

Moving in circles is done using the trigonometry functions Math.cos and Math.sin. For those who aren't familiar with these, I'll briefly introduce them since we will occasionally use them in this book.

Math.cos and Math.sin are useful for finding points that lie on a circle around point (0,0) with a radius of one. Both functions interpret their argument as the position on this circle, with zero denoting the point on the far right of the circle, going clockwise until 2π (about 6.28) has taken us around the whole circle. Math.cos tells you the x-coordinate of the point that corresponds to the given position, and Math.sin yields the y-coordinate. Positions (or angles) greater than 2π or less than 0 are valid—the rotation repeats so that a+2π refers to the same angle as a.

This unit for measuring angles is called radians—a full circle is 2π radians, similar to how it is 360 degrees when measuring in degrees. The constant π is available as Math.PI in JavaScript.

The cat animation code keeps a counter, angle, for the current angle of the animation and increments it every time the animate function is called. It can then use this angle to compute the current position of the image element. The top style is computed with Math.sin and multiplied by 20, which is the vertical radius of our ellipse. The left style is based on Math.cos and multiplied by 200 so that the ellipse is much wider than it is high.

Note that styles usually need *units*. In this case, we have to append "px" to the number to tell the browser that we are counting in pixels (as opposed to centimeters, “ems”, or other units). This is easy to forget. Using numbers without units will result in your style being ignored—unless the number is 0, which always means the same thing, regardless of its unit.