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, your browser retrieves the page's HTML text and parses it, much like 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.

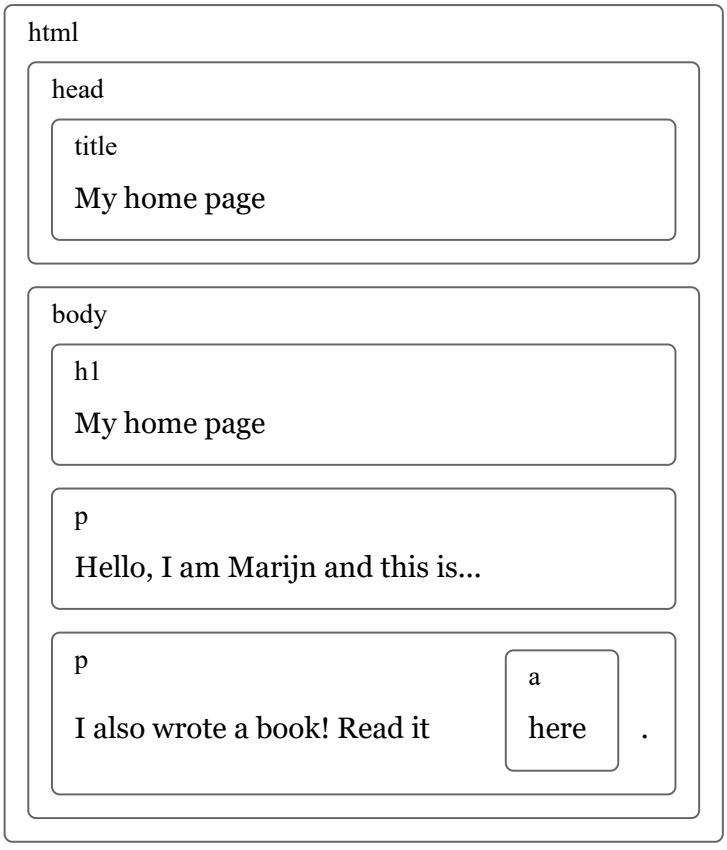
This representation of the document is one of the toys that a JavaScript program has available in its sandbox. It is a data structure that you can read or modify. It acts as a *live* data structure: when it's modified, the page on the screen is updated to reflect the changes.

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, no cycles (a node may not contain itself, directly or indirectly), and a single, well-defined *root*. In the case of the DOM, `document.documentElement` serves as the root.

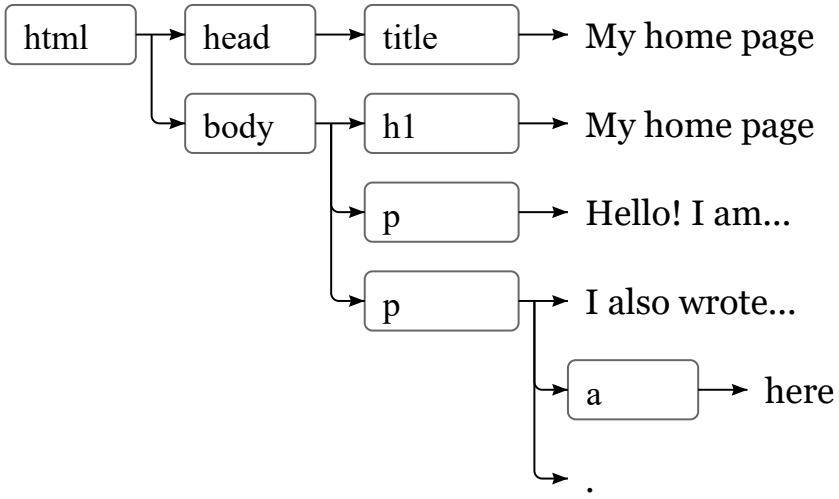
Trees come up a lot in computer science. In addition to representing recursive structures such as HTML documents or programs, they are often used to maintain sorted sets of data because elements can usually be found or inserted more efficiently in a tree than in a flat array.

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. This is because the DOM interface wasn't designed for JavaScript alone. 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're using will save you more time than having a familiar interface across languages.

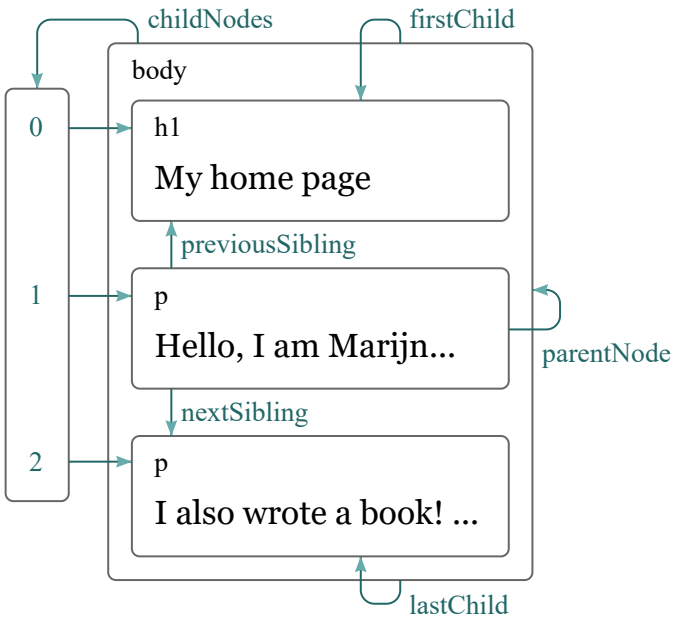
As an example of this poor integration, consider the `childNodes` property that element nodes in the DOM have. This property holds an array-like object with a `length` property and properties labeled by numbers to access the child nodes. But it is an instance of the `NodeList` type, not a real array,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

Then there are issues that are simply caused by 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 we 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 has not just three children (`<h1>` and two `<p>` elements), but seven: those three, plus the spaces before, after, and between them.

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.

```
<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.

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. To read or change custom attributes, which aren't available as regular object properties, you have to use the `getAttribute` and `setAttribute` methods.

```
<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"` with 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 `<strong>` 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);
  // → 19
  console.log("offsetHeight:", para.offsetHeight);
  // → 25
</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 pixel positions 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 before doing so. 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 that 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—`<strong>` 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 `<strong>` 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. For example, 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 very 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 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. It has a default 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. 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.

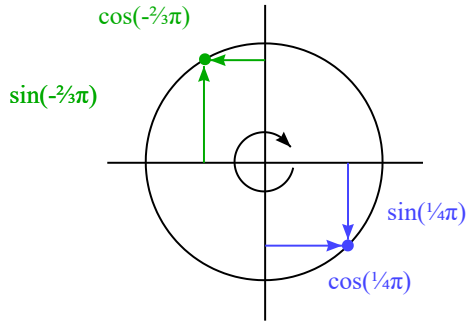
If we just updated the DOM in a loop, the page would freeze, and nothing would show up on the screen. Browsers do not update their display while a JavaScript program is running, nor do they allow any interaction with the page. This is why we need `requestAnimationFrame`—it lets the browser know that 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 when, for example, another heavy task running on the same computer prevented 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 1. Both functions interpret their argument as the position on this circle, with 0 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.

Summary

JavaScript programs may inspect and interfere with the document that the browser is displaying through a data structure called the DOM. This data structure represents the browser’s model of the document, and a JavaScript program can modify it to change the visible document.

The DOM is organized like a tree, where elements are arranged hierarchically according to the structure of the document. The objects representing elements have properties such as `parentNode` and `childNodes`, which can be used to navigate through this tree.

The way a document is displayed can be influenced by *styling*, both by attaching styles to nodes directly and by defining rules that match certain nodes. There are many different style properties, such as `color` or `display`. JavaScript code can manipulate an element’s style directly through its `style` property.

Exercises

Build a table

An HTML table is built with the following tag structure:

```
<table>
  <tr>
    <th>name</th>
    <th>height</th>
    <th>place</th>
  </tr>
  <tr>
    <td>Kilimanjaro</td>
    <td>5895</td>
    <td>Tanzania</td>
  </tr>
</table>
```

For each *row*, the `<table>` tag contains a `<tr>` tag. Inside of these `<tr>` tags, we can put cell elements: either heading cells (`<th>`) or regular cells (`<td>`).

Given a dataset of mountains, an array of objects with `name`, `height`, and `place` properties, generate the DOM structure for a table that enumerates the objects. It has one column per key and one row per object, plus a header row with `<th>` elements at the top, listing the column names.

Write this so that the columns are automatically derived from the objects, by taking the property names of the first object in the data.

Show the resulting table in the document by appending it to the element that has an `id` attribute of `"mountains"`.

Once you have this working, right-align cells that contain number values by setting their `style.textAlign` property to `"right"`.

```
<h1>Mountains</h1>

<div id="mountains"></div>

<script>
```

```
  const MOUNTAINS = [
    {name: "Kilimanjaro", height: 5895, place: "Tanzania"},
    {name: "Everest", height: 8848, place: "Nepal"},
    {name: "Mount Fuji", height: 3776, place: "Japan"},
    {name: "Vaalserberg", height: 323, place: "Netherlands"},
    {name: "Denali", height: 6168, place: "United States"},
    {name: "Popocatepetl", height: 5465, place: "Mexico"},
    {name: "Mont Blanc", height: 4808, place: "Italy/France"}
  ];

  // Your code here
</script>
```

▶ Display hints...

Elements by tag name

The `document.getElementsByTagName` method returns all child elements with a given tag name. Implement your own version of this as a function that takes a node and a string (the tag name) as arguments and returns an array containing all descendant element nodes with the given tag name. Your function should go through the document itself. It may not use a method like `querySelectorAll` to do the work.

To find the tag name of an element, use its `nodeName` property. But note that this will return the tag name in all uppercase. Use the `toLowerCase` or `toUpperCase` string methods to compensate for this.

```
<h1>Heading with a <span>span</span> element.</h1>
<p>A paragraph with <span>one</span>, <span>two</span>
  spans.</p>

<script>
  function byTagName(node, tagName) {
    // Your code here.
  }

  console.log(byTagName(document.body, "h1").length);
  // → 1
  console.log(byTagName(document.body, "span").length);
  // → 3
  let para = document.querySelector("p");
```

```
  console.log(byTagName(para, "span").length);
  // → 2
</script>
```

▶ Display hints...

The cat's hat

Extend the cat animation defined earlier so that both the cat and his hat (`<img src="img/hat.png">`) orbit at opposite sides of the ellipse.

Or make the hat circle around the cat. Or alter the animation in some other interesting way.

To make positioning multiple objects easier, you'll probably want to switch to absolute positioning. This means that `top` and `left` are counted relative to the top left of the document. To avoid using negative coordinates, which would cause the image to move outside of the visible page, you can add a fixed number of pixels to the position values.

```
<style>body { min-height: 200px }</style>
<img src="img/cat.png" id="cat" style="position: absolute">
<img src="img/hat.png" id="hat" style="position: absolute">

<script>
  let cat = document.querySelector("#cat");
  let hat = document.querySelector("#hat");

  let angle = 0;
  let lastTime = null;
  function animate(time) {
    if (lastTime != null) angle += (time - lastTime) * 0.001;
    lastTime = time;
    cat.style.top = (Math.sin(angle) * 40 + 40) + "px";
    cat.style.left = (Math.cos(angle) * 200 + 230) + "px";

    // Your extensions here.

    requestAnimationFrame(animate);
  }
  requestAnimationFrame(animate);
</script>
```

▶ Display hints...