O'REILLY®

Third Edition

JavaScript Cookbook

Programming the Web

Early Release
RAW & UNEDITED

John Paxton,
Adam D. Scott &
Shelley Powers

JavaScript Cookbook

THIRD EDITION

John Paxton, Adam D. Scott, and Shelley Powers

JavaScript Cookbook, 3E

By John Paxton, Adam D. Scott, and Shelley Powers

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

- Editors: Simon St. Laurent and Brian MacDonald
- Production Editor: Kara Ebrahim
- Copyeditor: Jasmine Kwityn
- Proofreader: Kara Ebrahim
- Indexer: Judy McConville
- Interior Designer: David Futato
- Cover Designer: Ellie Volckhausen
- Illustrator: Rebecca Demarest

- July 2010: First Edition
- February 2015: Second Edition
- November 2020: Third Edition

Revision History for the Early Release

- 2020-06-03: First release

See http://oreilly.com/catalog/errata.csp?isbn=9781491901885 for release details.

978-1-492-05568-6

Chapter 1. Errors

Errors. Everyone has them. No one is perfect. Things go wrong. Sometimes the error could have been anticipated, other times, not so much. To modify a cliche, it's not so much the error as what you do with it that matters.

What should we do with our errors, then? The default behavior is for JavaScript to die at the point of the error, exiting with a stack trace. We can capture an error, react to it, modify it, re-throw it, even hide it if we choose. Returning to a theme of this cookbook: just because we can, does not mean we should. Our guiding light is effective, efficient, reusable JavaScript programming. Let's look at Errors under this light.

JavaScript is not robust at error handing in general. The language has the Error object, which is flexible enough. But catching errors with `try-catch` lacks features found in other languages. We cannot catch errors by their type. We cannot (easily) filter by error. Further, in the browser, it is rare that we can recover from an error. Rare enough that the recipes below deal with the few common error situations from which an application can recover. In Node.js, throwing errors causes enough problems that an entirely different pattern of error handling involed. We will look at Node's peculiar form of error "handling" as well.

Our goal should be to make our programming experience better. Instead of our JavaScript engine dying horribly with an incomprehensible error, we can add and modify information about the error, illuminating the path for the person (likely ourselves) tasked with fixing the error. Allow the failure to happen, but clarify why the error occurred and who raised it.

1.1 Using Errors

JavaScript has eight Error types. The parent type is the aptly-named `Error`. There are seven subtypes, which we will look at in the next recipe. What does an Error give us? We can count on three properties: a constructor, `name`, and `message`. These are all available to subclasses of Error as well. We also have access to a `toString()` method which will usually return the `message` property. If we are using the V8 JavaScript engine (Chrome, Node.js, possibly others), we can use `captureStackTrace()` as well. More on this soon. Let's start with a standard JavaScript Error object.

Problem

You want to create, throw, and catch a standard error

Solution

Create an instance of Error, or a subtype, throw it, and catch it later on. Perhaps that is too brief a description. Start by creating an instance of Error. The constructor takes a string as an argument. Pass something useful and indicative of what the problem was. Consider keeping a list of the various error strings used in the application so that they are consistent and informative. Use the `throw` keyword to throw the instance you created. Then catch it in a `try-catch` block.

```
function willThrowError() {
  if ( /* something goes wrong */) {
    // Create an error with useful information.
    // Don't be afraid to duplicate something from the stack
trace, like the method name
    throw new Error(`Problem in ${method}, ${reason}`);
  }
}

// Somewhere else in your code

try {
  willThrowError();
  // Other code will not be reached
} catch (error) {
  console.error('There was an error: ', error);
}
```

Discussion

We can create an error either with the `new` keyword before the Error or not. If `Error()` is called without `new` preceding it, `Error()` acts as a function which returns an Error object. The end result is the same. The Error constructor takes one standard argument: the error message. This will be assigned to the error’s `message` property. Some engines allow for additional non-standard arguments to the constructor, but these are not part of a current ECMAScript spec and are not on track to be on a future one. Wrapping the code that will throw an error in a `try-catch` block allows us to catch the error within our code. Had we not done so, the error would have propagated to the top of the stack and exited JavaScript. Here, we are reporting the error to the console with `console.error`. This is more or less the default behavior, though we can add information as part of the call to `console.error.`

1.2 Capturing Errors by their subtypes

JavaScript has seven subtypes of Error. We can check for a subtype of error to determine the kind of error raised by a problem in our code. This may illuminate the possibility of recovery, or at least give us a little more information about what went wrong.

The seven Error subtypes:

- EvalError: thrown by use of the built-in function eval()
- InternalError: Internal to the JavaScript engine; not part of the ECMAScript spec, but used by engines for non-standard errors
- RangeError: A value is outside of its valid range
- ReferenceError: Raised when encountering a problem trying to dereference an invalid reference
- SyntaxError: A problem with the syntax of evaluated code, including JSON

- TypeError: A variable or parameter is of an unexpected or wrong type
- URIError: Raised by problems with `encodeURI()` and `decodeURI()`

Problem

How do we catcn errors by their subtype?

Solution

In your catch block, check the specific error type

```
try {
  // Some code that will raise an error
} catch (err) {
  if (err instanceof RangeError) {
    // Do something about the value being out of range
  } else if (err instanceof TypeError) {
    // Do something about the value being the wrong type
  } else {
    // Rethrow the error
    throw err;
  }
}
```

Discussion

We may need to respond to specific subtypes of Error. Perhaps the subtype can tell us something about what went wrong with the code. Or maybe our code threw a specific Error subtype on purpose, to carry additional information about the problem in our code. Both TypeError and RangeError lend themselves to this behavior, for example. The problem is that JavaScript permits only one catch block, offering neither opportunity nor syntax to capture errors by their type. Given JavaScript's origins, this is not surprising, but it is nonetheless inconventient.

Our best option is to check the type of the Error ourselves. Use an `if` statement with the `instanceof` operator to check the type of the caught error. Remember to write a complete if-else statement, otherwise your code may accidentally swallow the error and suppress it. In the code above, we assume that we cannot do anything useful with error types that are neither RangeErrors nor TypeErrors. We re-throw other errors, so that code higher up the stack than this can choose to capture the error. Or, as is appropriate, the error could bubble to the top of the stack, as it normally would.

We should note: handling errors, even by their subtype, is fraught with difficulties in JavaScript. With rare exceptions, an error raised by your JavaScript engine cannot be recovered from in a meaningful sense. It is better to try to bring useful information about the error to the attention of the user, rather than to try to bring JavaScript back into a correct state (which may, in fact, be impossible!).

For example: Consider an EvalError. What are we, as coders, going to do if there's an EvalError or a SyntaxEror? It seems that we should go an fix our code. What about a ReferenceError? Code cannot recover from trying to dereference an invalid referent. We might be able to write a catch block that returns something more informative than "Attempted to call getFoo() on undefined". But we cannot determine what the original code intended. We can only repackage the error and exit gracefully (if possible). In the next section, we will look at two error types that offer the possibility of recovery, as well as throwing our own, more useful error types.

1.3 Throwing useful errors

Given the limits of JavaScript and error handling, are there Error subtypes worth using? Several of the subtypes of Error are limited to very specific cases. But two, the RangeError, and the TypeError show some promise.

Problem

You want to throw useful Error subtypes

Solution

Use TypeError to express an incorrect parameter or variable type

```
function calculateValue(x) {
  if (typeof x !== 'number') {
    throw new TypeError(`Value [${x}] is not a number.`);
  }

  // Rest of the function
}
```

Use RangeError to express that a parameter or value is out of range

```
function setAge(age) {
  const upper = 125;
  const lower = 18;
  if (age > 125 || age < 18) {
    throw new RangeError(`Age [${age}] is out of the
acceptable range of ${lower} to ${upper}.`);
  }
}
```

Discussion

If you are going to use Errors to express incorrect states for your application, the TypeError and RangeError hold some promise. TypeError covers issues where the expected value was of the wrong type. What constitutes a "wrong" type? That's up to us as the designers. Our code might expect one of JavaScript's "primitive" values returned by a call to `typeof`. If our function receives an unantipated value type, we could throw a TypeError. We can say the same with `instanceof` for object types. These are not hard limits on when we might throw a TypeError, but they are good guidelines.

Going further, if our function received the right kind of value but it was outside of an acceptable range, we can throw a RangeError. Note that RangeErrors are specifically bound to numeric values. Put another way, we should not throw a RangeError when expecting a string and receiving one that is too short or too long. In the case of either RangeErrors or TypeErrors, make sure your error message is informative. Include the given value and information about the expected value.

Do not use errors for to validate forms or other input. Errors often do not show up in the HTML of a page, as they are re-routed to the console. Even if they do show up in the visible part of the page, they often contain impenetrable and weird content which will confuse and annoy the common user. And if they are not anticipated as part of your rendered content, they can throw off the rendering of the rest of the view. Invalid user data should be an expected state of your code, and should be handled in a user-friendly manner. There are both native APIs for form validation, as well as customizable, event-based hooks for validation. Use those instead. Errors in JavaScript indicate a state where your code received input that is just wrong and should be corrected.

1.4 Throwing custom errors

The Error type might be too broad. Most subclasses are too specific, or too limiting. What if we want to create our own Error types? How should we subclass Error?

Problem

How do we create our own Error subtypes?

Solution

Create a subtype of Error by subclassing Error using the ECMAScript 2015 class inheritance syntax

```
class CustomError extends Error {
  constructor(customProp='customValue', ...params) {
    super(...params);

    if (Error.captureStackTrace) {
      Error.captureStackTrace(this, CustomError);
    }
    this.name = 'CustomError';
    this.customProp = customProp;
  }
}
```

Discussion

When subclassing Error, we should keep in mind two possibly competing concerns: stayign within the bounds of a typical JavaScript error, and expressing enough information for our customized error. In the former case, do not attempt to recreate the errors or exceptions of your second favorite language. Do not over-extend JavaScript's Error type with extra methods and properties. A later programmer using your new Error subtype should be able to make reasonable assumptions about the subtype's API. They should be able to access standard information in the standard way.

Your subclass constructor can take custom properties, which should be distinct from the standard argument of an error message. Keep in mind that Firefox and some other browsers expect the first three arguments to the Error constructor to be a message string, a filename, and a line number. If your subclass's constructor takes custom arguments, add the to the front of the parameters list. This will make capturing standard arguments easier, as we can use ECMAScript 2015's rest parameter feature to vaccuum up any remaining arguments.

In the constructor for your custom error, call `super()` first, passing any standard parameters. Because your code might run on V8 (either Chrome or Node.js), properly set this Error's stack trace. Check for the `captureStackTrace` method, and, if present, call Error.captureStackTrace passing it a reference to the current instance (as `this`) and your CustomError class.

Set the `name` property of your custom error subclass. This ensures that, when reporting to the console, your error will carry the name of your subclass, instead of the more generic Error class.

Set any custom properties as necessary.

1.5 Handling JSON parsing errors

JavaScript Object Notation has become a popular, portable data format thanks in part to the popularity of JavaScript. There are two native functions availabe for JSON processing: for JavaScript-to-JSON conversion, use `JSON.stringify(jsObject)`. To deserialize a JSON string into a JavaScript object (or string, or array, etc.) use `JSON.parse(jsonString)`. Conversion of JavaScript to a JSON string should not raise any errors, barring issues with the JavaScript engine implementing `JSON.stringify`. On the other hand, converting a JSON string into JavaScript has many potential issues. The JSON could be malformed, or could just not be JSON at all! JavaScript raises error with JSON parsing as SyntaxErrors, which is sub-optimal. A SyntaxError is raised when JavaScript tries to interpret syntactically invalid code. While technically accurate, this lacks detail in the case of JSON parsing.

Problem

How should we handle parsing of bad or malformed JSON data?

Solution

Create a custom subtype of SyntaxError which is raised on issues with JSON parsing.

```
class JSONParseError extends SyntaxError {
  constructor(...params) {
    super(...params);

    if (Error.captureStackTrace) {
      Error.captureStackTrace(this, JSONParseError);
    }

    this.name = 'JSONParseError';
  }
}

function betterParse(jsonString) {
  try {
    JSON.parse(jsonString);
  } catch (err) {
    if (err instanceof SyntaxError) {
      throw new JSONParseError(err);
    } else {
      throw err;
    }
  }
}
```

Discussion

We would like to log a specific error type when `JSON.parse` encounters an error. We can create a class, `JSONParseError`, which is a subclass of `SyntaxError`, for problems with JSON parsing. The class itself does not do much, other than establishing the name and type `JSONParseError`. But consumers of this API can now test for JSONParseError as opposed to the too-general SyntaxError. Logging will improve as well.

Remembering to re-throw SyntaxErrors as JSONParseErrors anytime we parse JSON strings is tedious. Instead, we will call `betterParse(jsonString)` which does the work for us. The `betterParse` function wraps the call to `JSON.parse()` in a `try` block, safely catching any SyntaxErrors. If a SyntaxError is raised, we will re-package it as a JSONParseError. If some other error is raised (it could happen), we will pass that error along as-is, with no modification or re-packaging.

Chapter 2. Working with HTML

In 1995 Netscape tasked software developer Brendan Eich with creating a programming language designed for adding interactivity to pages in the Netscap Navigator browser. In response, Eich infamously developed the first version of JavaScript in 10 days. A few years later, JavaScript has became a cross-browser standard through the adoption of the ECMAScript standardization.

Despite the early attempt at standardization, web developers battled for years with browsers that had different JavaScript engine interpretations or features. Popular libraries, such as jQuery effectively allowed us to write simple cross-browser JavaScript. Thankfully, today's browsers share a near uniform implementation of the language. Allowing web developers to write "vanilla" (library-free) JavaScript to interact with an HTML page.

When working with HTML, we are working with the Document Object Model (DOM), which is the data representation of the HTML page. The recipes in this chapter will review how to interact with the DOM of an HTML page by selecting, updating, and removing elements from the page.

2.1 Accessing a Given Element and Finding Its Parent and Child Elements

Problem

You want to access a specific web page element, and then find its parent and child elements.

Solution

Give the element a unique identifier:

```
<div id="demodiv">
  <p>
    This is text.
  </p>
</div>
```

Use `document.getElementById()` to get a reference to the specific element:

```
const demodiv = document.getElementById("demodiv");
```

Find its parent via the `parentNode` property:

```
const parent = demodiv.parentNode;
```

Find its children via the `childNodes` property:

```
const children = demodiv.childNodes;
```

Discussion

A web document is organized like an upside-down tree, with the topmost element at the root and all other elements branching out beneath. Except for the root element (HTML), each element has a parent `node`, and all of the elements are accessible via the `document`.

There are several different techniques available for accessing these document elements, or *nodes* as they're called in the Document Object Model (DOM). Today, we access these nodes through standardized versions of the DOM, such as the DOM Levels 2 and 3. Originally, though, a de facto technique was to access the elements through the browser object model, sometimes referred to as DOM Level 0. The DOM Level 0 was invented by the leading browser company of the time, Netscape, and its use has been supported (more or less) in most browsers since. The key object for accessing web page elements in the DOM Level 0 is the `document` object.

The most commonly used DOM method is `document.getElementById()`. It takes one parameter: a case-sensitive string with the element's identifier. It returns an `element` object, which is referenced to the element if it exists; otherwise, it returns null.

NOTE

There are numerous ways to get one specific web page element, including the use of selectors, covered later in the chapter. But you'll always want to use the most restrictive method possible, and you can't get more restrictive than `document.getElementById()`.

The returned `element` object has a set of methods and properties, including several inherited from the `node` object. The `node` methods are primarily associated with traversing the document tree. For instance, to find the parent node for the element, use the following:

```
const parent =
document.getElementById("demodiv").parentNode;
```

You can find out the type of element for each node through the `nodeName` property:

```
const type = parent.nodeName;
```

If you want to find out what children an element has, you can traverse a collection of them via a NodeList, obtained using the `childNodes` property:

```
let outputString = '';

if (demodiv.hasChildNodes()) {
  const children = demodiv.childNodes;
  children.forEach(child => {
    outputString += `has child ${child.nodeName} `;
  });
}
console.log(outputString);;
```

Given the element in the solution, the output would be:

```
"has child #text has child P has child #text "
```

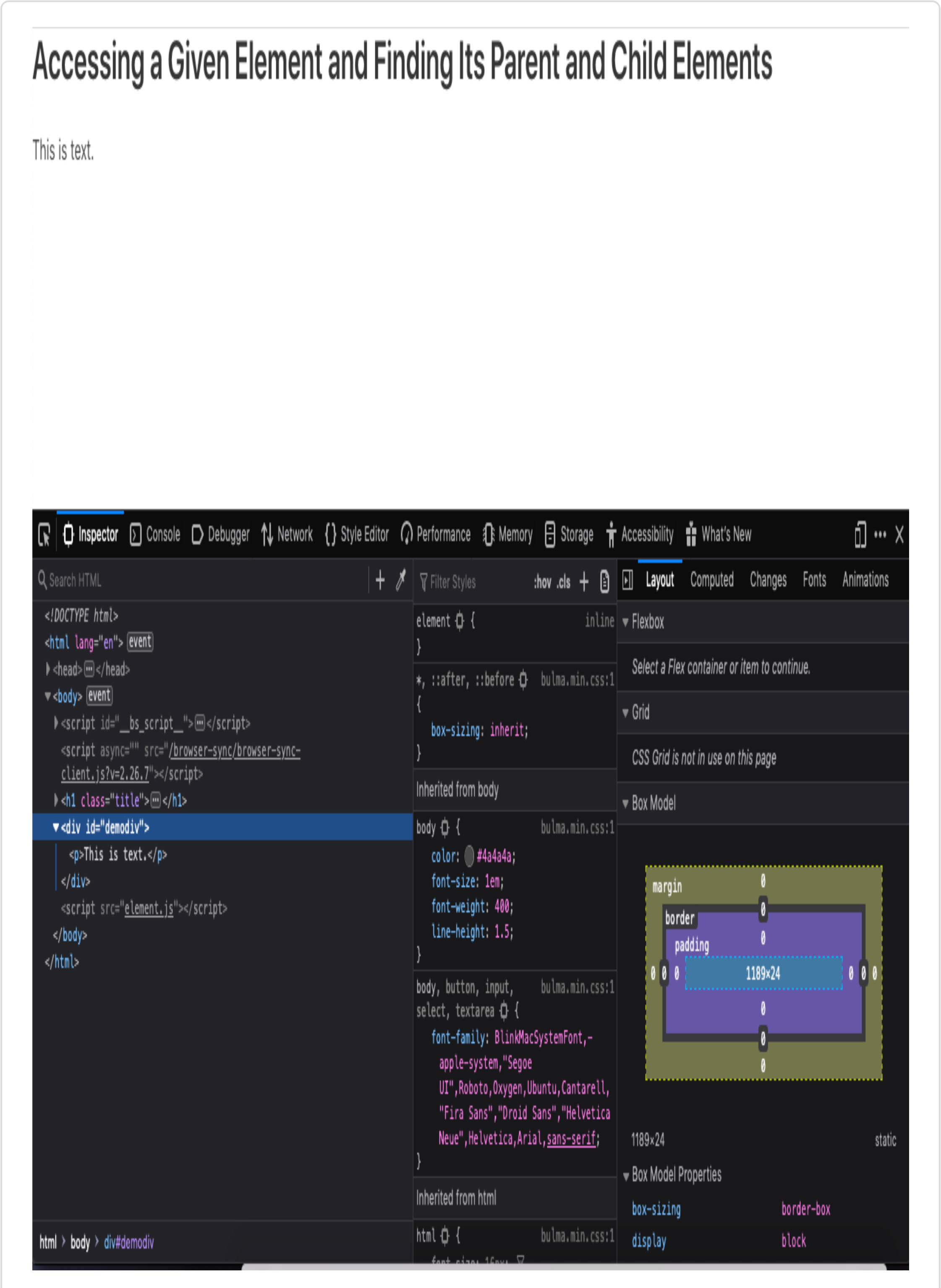
You might be surprised by what appeared as a child node. In this example, whitespace before and after the paragraph element is itself a child node with a `nodeName` of `#text`. For the following `div` element:

```
<div id="demodiv" class="demo">
  <p>Some text</p>
  <p>Some more text</p>
</div>
```

the `demodiv` element (node) has five children, not two:

```
has child #text
has child P
has child #text
has child P
has child #text
```

The best way to see how messy the DOM can be is to use a debugger such as the Firefox or Chrome developer tools, access a web page, and then utilize whatever DOM inspection tool the debugger provides. I opened a simple page in Firefox and used the developer tools to display the element tree, as shown in Figure 2-1.



Accessing a Given Element and Finding Its Parent and Child Elements

This is text.

```
<!DOCTYPE html>
<html lang="en"> event
  <head>…</head>
  <body> event
    <script id="__bs_script__">…</script>
    <script async="" src="/browser-sync/browser-sync-
    client.js?v=2.26.7"></script>
    <h1 class="title">…</h1>
    <div id="demodiv">
      <p>This is text.</p>
    </div>
    <script src="element.js"></script>
  </body>
</html>
```

html › body › div#demodiv

```
element {                          inline
}
*, ::after, ::before {       bulma.min.css:1
  box-sizing: inherit;
}
Inherited from body
body {                       bulma.min.css:1
  color: #4a4a4a;
  font-size: 1em;
  font-weight: 400;
  line-height: 1.5;
}
body, button, input,         bulma.min.css:1
select, textarea {
  font-family: BlinkMacSystemFont,-
    apple-system,"Segoe
    UI",Roboto,Oxygen,Ubuntu,Cantarell,
    "Fira Sans","Droid Sans","Helvetica
    Neue",Helvetica,Arial,sans-serif;
}
Inherited from html
html {                       bulma.min.css:1
```

Flexbox

Select a Flex container or item to continue.

Grid

CSS Grid is not in use on this page

Box Model

margin 0, border 0, padding 0, 1189×24

1189×24 static

Box Model Properties

box-sizing	border-box
display	block

Figure 2-1. Examining the element tree of a web page using Firefox's developer tools

2.2 Traversing the Results from querySelectorAll() with forEach()

Problem

You want to loop over the `nodeList` returned from a call to `querySelectorAll()`.

Solution

In modern browsers, you can use `forEach()` when working with a NodeList (the collection returned by `querySelectorAll()`):

```
// use querySelector to find all list items on a page
const items = document.querySelectorAll('li');

items.forEach(item => {
  console.log(item.firstChild.data);
});
```

Discussion

`forEach()` is an Array method, but the results of `querySelectorAll()` is a NodeList which is a different type of object than an Array. Thankfully, modern browsers have built in support for `forEach`, allowing us to iterate over a NodeList like an array.

Unfortunately, Internet Explorer (IE) does not support using `forEach` in this way. If you'd like to support IE, the recommended approach is to include a polyfill that uses a standard `for` loop under the hood:

```
if (window.NodeList && !NodeList.prototype.forEach) {
  NodeList.prototype.forEach = function(callback, thisArg) {
    thisArg = thisArg || window;
    for (var i = 0; i < this.length; i++) {
      callback.call(thisArg, this[i], i, this);
    }
  };
}
```

In the polyfill, we check for the existence of `Nodelist.prototype.forEach`. If it does not exist, a `forEach` method is added to the `Nodelist` prototype that uses a `for` loop to iterate over the results of a DOM query. By doing so, you can use the `forEach` syntax freely across your codebase.

2.3 Adding Up Values in an HTML Table

2.4 Problem

You want to sum all numbers in a table column.

Solution

Traverse the table column containing numeric string values, convert to numbers, and sum the numbers:

```
let sum = 0;

// use querySelector to find all second table cells
```

```
const cells = document.querySelectorAll('td:nth-of-
type(2)');

// iterate over each
cells.forEach(cell => {
  sum += Number.parseFloat(cell.firstChild.data);
});
```

Discussion

The `parseInt()` and `parseFloat()` methods convert strings to numbers, but `parseFloat()` is more adaptable when it comes to handling numbers in an HTML table. Unless you're absolutely certain all of the numbers will be integers, `parseFloat()` can work with both integers and floating-point numbers.

As you traverse the HTML table and convert the table entries to numbers, sum the results. Once you have the sum, you can use it in a database update, pop up a message box, or print it to the page, as the solution demonstrates.

You can also add a sum row to the HTML table. Example 2-1 demonstrates how to convert and sum up numeric values in an HTML table, and then how to insert a table row with this sum, at the end. The code uses `document.querySelectorAll()`, which uses a different variation on the CSS selector, `td + td`, to access the data this time. This selector finds all table cells that are preceded by another table cell.

Example 2-1. Converting table values to numbers and summing the results

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Adding Up Values in an HTML Table</title>
</head>
<body>
  <h1>Adding Up Values in an HTML Table</h1>
    <table>
      <tbody id="table1">
        <tr>
            <td>Washington</td><td>145</td>
        </tr>
        <tr>
            <td>Oregon</td><td>233</td>
        </tr>
        <tr>
            <td>Missouri</td><td>833</td>
        </tr>
      <tbody>
    </table>

    <script>
      let sum = 0;

      // use querySelector to find all second table cells
      const cells = document.querySelectorAll('td + td');

      // iterate over each
      cells.forEach(cell => {
        sum += Number.parseFloat(cell.firstChild.data);
      });

      // now add sum to end of table
      const newRow = document.createElement('tr');
      // first cell
      const firstCell = document.createElement('td');
      const firstCellText = document.createTextNode('Sum:');
      firstCell.appendChild(firstCellText);
      newRow.appendChild(firstCell);

      // second cell with sum
      const secondCell = document.createElement('td');
      const secondCellText = document.createTextNode(sum);
      secondCell.appendChild(secondCellText);
      newRow.appendChild(secondCell);

      // add row to table
      document.getElementById('table1').appendChild(newRow);
    </script>
</body>
</html>
```

Being able to provide a sum or other operation on table data is helpful if you're working with dynamic updates via an Ajax operation, such as accessing rows of data from a database. The Ajax operation may not be able to provide summary data, or you may not want to provide summary data until a web page reader chooses to do so. The users may want to manipulate the table results, and then push a button to perform the summing operation.

Adding rows to a table is straightforward, as long as you remember the steps:

- Create a new table row using `document.createElement("tr")`.
- Create each table row cell using `document.createElement("td")`.

- Create each table row cell’s data using `document.createTextNode()`, passing in the text of the node (including numbers, which are automatically converted to a string).
- Append the text node to the table cell.
- Append the table cell to the table row.
- Append the table row to the table. Rinse, repeat.

Extra: forEach and querySelectorAll

In the above example, I’m using the `forEach()` method to iterate over the results of `querySelectorAll()`, which returns a NodeList, not an array. Though `forEach()` is an array method, modern browsers have implemented `NodeList.prototype.forEach()`, which enables it iterating over a NodeList with the `forEach()` syntax. The alternative would be a loop:

```
let sum = 0;

// use querySelector to find all second table cells
let cells = document.querySelectorAll("td:nth-of-type(2)");

for (var i = 0; i < cells.length; i++) {
  sum+=parseFloat(cells[i].firstChild.data);
}
```

Extra: Modularization of Globals

```
As part of a growing effort to _modularize_ JavaScript, the
+parseFloat()+ and +parseInt()+ methods are now attached to
```

```
the Number object, as new _static_ methods, as of ECMAScript
2015:
```

```
// modular method
const modular = Number.parseInt('123');
// global method
const global = parseInt('123');
```

These modules have reached widespread browser adoption, but can be polyfilled for older browser support, using a tool like Babel, or on their own:

```
if (Number.parseInt === undefined) {
  Number.parseInt = window.parseInt
}
```

2.5 Finding All Elements That Share an Attribute

Problem

You want to find all elements in a web document that share the same attribute.

Solution

Use the *universal selector* (`*`) in combination with the attribute selector to find all elements that have an attribute, regardless of its value:

```
const elems = document.querySelectorAll('*[class]');
```

The universal selector can also be used to find all elements with an attribute that’s assigned the same value:

```
const reds = document.querySelectorAll('*[class="red"]');
```

Discussion

The solution demonstrates a rather elegant query selector, the *universal selector* (`*`). The universal selector evaluates all elements, so it’s the one you want to use when you need to verify *something* about each element. In the solution, we want to find all of the elements with a given attribute.

To test whether an attribute exists, all you need to do is list the attribute name within square brackets (*`[attrname]`*). In the solution, we’re first testing whether the element contains the `class` attribute. If it does, it’s returned with the element collection:

```
var elems = document.querySelectorAll('*[class]');
```

Next, we’re getting all elements with a `class` attribute value of *`red`*. If you’re not sure of the class name, you can use the substring-matching query selector:

```
const reds = document.querySelectorAll('*[class="red"]');
```

Now any class name that contains the substring “red” matches.

You could also modify the syntax to find all elements that don’t have a certain value. For instance, to find all `div` elements that don’t have the target class name, use the `:not` negation operator:

```
const notRed = document.querySelectorAll('div:not(.red)');
```

2.6 Accessing All Images in a Page

Problem

You want to access all `img` elements in a given document.

Solution

Use the `document.getElementsByTagName()` method, passing in *`img`* as the parameter:

```
const imgElements = document.getElementsByTagName('img');
```

Discussion

The `document.getElementsByTagName()` method returns a collection of nodes (a NodeList) of a given element type, such as the `img` tag in the solution. The collection can be traversed like an array, and the order of nodes is based on the order of the elements within the document (the first `img` element in the page is accessible at index 0, etc.):

```
const imgElements = document.getElementsByTagName('img');
for (let i = 0; i < imgElements.length; i += 1) {
  const img = imgElements[i];
  ...
}
```

The `NodeList` collection can be traversed like an array, but it isn't an `Array` object—you can't use `Array` object methods, such as `push()` and `reverse()`, with a `NodeList`. Its only property is `length`, and its only method is `item()`, returning the element at the position given by an index passed in as parameter:

```
const img = imgElements.item(1); // second image
```

`NodeList` is an intriguing object because it's a live collection, which means changes made to the document after the `NodeList` is retrieved are reflected in the collection. Example 2-2 demonstrates the `NodeList` live collection functionality, as well as `getElementsByTagName`.

In the example, three images in the web page are accessed as a `NodeList` collection using the `getElementsByTagName` method. The `length` property, with a value of `3`, is output to the console. Immediately after, a new paragraph and `img` elements are created, and the `img` is appended to the paragraph. To append the paragraph following the others in the page, `getElementsByTagName` is used again, this time with the paragraph tags (`p`). We're not really interested in the paragraphs, but in the paragraphs' parent elements, found via the `parentNode` property on each paragraph.

The new paragraph element is appended to the paragraph's parent element, and the previously accessed `NodeList` collection's length property is again printed out. Now, the value is `4`, reflecting the addition of the new `img` element.

Example 2-2. Demonstrating getElementsByTagName and the NodeList live collection property

```html
<!DOCTYPE html>
<html>
<head>
<title>NodeList</title>
</head>
<body>
  <p><img src="firstimage.jpg" alt="image description" /></p>
  <p><img src="secondimage.jpg" alt="image description" /></p>
  <p><img src="thirdimage.jpg" alt="image description" /></p>

<script>
  const imgs = document.getElementsByTagName('img');
  console.log(imgs.length);
  const p = document.createElement('p');
  const img = document.createElement('img');
  img.src = './img/someimg.jpg';
  p.appendChild(img);

  const paras = document.getElementsByTagName('p');
  paras[0].parentNode.appendChild(p);

  console.log(imgs.length);
</script>

</body>
</html>
```

In addition to using `getElementsByTagName()` with a specific element type, you can also pass the universal selector (`*`) as a parameter to the method to get all elements:

```
const allElems = document.getElementsByTagName('*');
```

See Also

In the code demonstrated in the discussion, the children nodes are traversed using a traditional `for` loop. In modern browsers the `forEach()`, method be used directly with a NodeList, as is demonstrated in Recipe 2.2.

2.7 Discovering All Images in Articles Using the Selectors API

Problem

You want to get a list of all `img` elements that are descendants of `article` elements, without having to traverse an entire collection of elements.

Solution

Use the Selectors API and access the `img` elements contained within `article` elements using CSS-style selector strings:

```
const imgs = document.querySelectorAll('article img');
```

Discussion

There are two selector query API methods. The first, `querySelectorAll()`, is demonstrated in the solution; the second is `querySelector()`. The difference between the two is querySelectorAll(), which returns all elements that match the selector criteria, while querySelector() only returns the first found result.

The selectors syntax is derived from CSS selector syntax, except that rather than style the selected elements, they're returned to the application. In the example, all img elements that are descendants of article elements are returned. To access all img elements regardless of parent element, use:

```
const imgs = document.querySelectorAll('img');
```

In the solution, you'll get all img elements that are direct or indirect descendants of an article element. This means that if the img element is contained within a div that's within an article, the img element will be among those returned:

```
<article>
   <div>
      <img src="..." />
   </div>
</article>
```

If you want only those img elements that are direct children of an article element, use the following:

```
const imgs = document.querySelectorAll('article> img');
```

If you're interested in accessing all img elements that are immediately followed by a paragraph, use:

```
const imgs = document.querySelectorAll('img + p');
```

If you’re interested in an `img` element that has an empty `alt` attribute, use the following:

```
const imgs = document.querySelectorAll('img[alt=""]');
```

If you’re only interested in `img` elements that don’t have an empty `alt` attribute, use:

```
const imgs = document.querySelectorAll('img:not([alt=""])');
```

The negation pseudoselector (`:not`) is used to find all `img` elements with `alt` attributes that are not empty.

Unlike the collection returned with `getElementsByTagName()` covered earlier, the collection of elements returned from `querySelectorAll()` is *not* a “live” collection. Updates to the page are not reflected in the collection if the updates occur after the collection is retrieved.

NOTE

Though the Selectors API is a wonderful creation, it shouldn’t be used for every document query. You should always use the most restrictive query when accessing elements. For instance, it’s more efficient to use `getElementById()` to get one specific element given an identifier.

See Also

There are three different CSS selector specifications, labeled as Selectors Level 1, Level 2, and Level 3. CSS Selectors Level 3 contains links to the documents defining the other levels. These documents provide the definitions of, and examples for, the different types of selectors.

2.8 Setting an Element’s Style Attribute

Problem

You want to add or replace a style setting on a specific web page element.

Solution

To change one CSS property, modify the property value via the element’s `style` property:

```
elem.style.backgroundColor = 'red';
```

To modify one or more CSS properties for a single element, you can use `setAttribute()` and create an entire CSS style rule:

```
elem.setAttribute('style',
  'background-color: red; color: white; border: 1px solid
black');
```

Or you can predefine the style rule, assign it a class name, and set the `class` property for the element:

```
.stripe{
  background-color: red;
  color: white;
  border: 1px solid black;
}

...

elem.setAttribute('class', 'stripe');
```

Discussion

An element’s CSS properties can be modified in JavaScript using one of three approaches. As the solution demonstrates, the simplest approach is to set the property’s value directly using the element’s `style` property:

```
elem.style.width = '500px';
```

If the CSS property contains a hyphen, such as `font-family` or `background-color`, use the *CamelCase notation* for the property:

```
elem.style.fontFamily = 'Courier';
elem.style.backgroundColor = 'rgb(255,0,0)';
```

The CamelCase notation removes the dash, and capitalizes the first letter following the dash.

You can also use `setAttribute()` to set the `style` property:

```
elem.setAttribute('style','font-family: Courier; background-
color: yellow');
```

The `setAttribute()` method is a way of adding an attribute or replacing the value of an existing attribute for a web page element. The first argument to the method is the attribute name (automatically lowercased if the element is an HTML element), and the new attribute value.

When setting the `style` attribute, all CSS properties that are changed settings must be specified at the same time, as setting the attribute erases any previously set values. However, setting the `style` attribute using `setAttribute()` does not erase any settings made in a stylesheet, or set by default by the browser.

A third approach to changing the style setting for the element is to modify the `class` attribute:

```
elem.setAttribute('class', 'classname');
```

Advanced

Rather than using `setAttribute()` to add or modify the attribute, you can create an attribute and attach it to the element using `createAttribute()` to create an `Attr` node, set its value using the `nodeValue` property, and then use `setAttribute()` to add the attribute to the element:

```
const styleAttr = document.createAttribute('style');
styleAttr.nodeValue = 'background-color: red';
someElement.setAttribute(styleAttr);
```

You can add any number of attributes to an element using either `createAttribute()` and `setAttribute()`, or `setAttribute()` directly. Both approaches are equally efficient, so unless there's a real need, you'll most likely want to use the simpler approach of setting the attribute name and value directly using `setAttribute()`.

When would you use `createAttribute()`? If the attribute value is going to be another entity reference, as is allowed with XML, you'll need to use `createAttribute()` to create an `Attr` node, as `setAttribute()` only supports simple strings.

Extra: Accessing an Existing Style Setting

For the most part, accessing existing attribute values is as easy as setting them. Instead of using `setAttribute()`, use `getAttribute()`:

```
const className =
document.getElementById('elem1').getAttribute('class');
```

Getting access to a style setting, though, is much trickier, because a specific element's style settings at any one time is a composite of all settings merged into a whole. This *computed style* for an element is what you're most likely interested in when you want to see specific style settings for the element at any point in time. Happily, there is a method for that: `window.getComputedStyle()`, which will return the current computed styles applied to the element.

```
const style = window.getComputedStyle(elem);
```

2.9 Inserting a New Paragraph

Problem

You want to insert a new paragraph just before the third paragraph within a `div` element.

Solution

Use some method to access the third paragraph, such as `getElementsByTagName()`, to get all of the paragraphs for a `div` element. Then use the `createElement()` and `insertBefore()` DOM methods to add the new paragraph just before the existing third paragraph:

```
// get the target div
const div = document.getElementById('target');

// retrieve a collection of  paragraphs
const paras = div.getElementsByTagName('p');

// create the element and append text to it
const newPara = document.createElement('p');
const text = document.createTextNode('New paragraph
content');
newPara.appendChild(text);

// if a third para exists, insert the new element before
// otherwise, append the paragraph to the end of the div
if (paras[2]) {
  div.insertBefore(newPara, paras[2]);
} else {
  div.appendChild(newPara);
}
```

Discussion

The `document.createElement()` method creates any HTML element, which then can be inserted or appended into the page. In the solution, the new paragraph element is inserted before an existing paragraph using `insertBefore()`.

Because we’re interested in inserting the new paragraph before the existing third paragraph, we need to retrieve a collection of the `div` element’s paragraphs, check to make sure a third paragraph exists, and then use `insertBefore()` to insert the new paragraph before the existing one. If the third paragraph doesn’t exist, we can append the element to the end of the `div` element using `appendChild()`.

About the Author

John Paxton has been programming in, consulting about, and training students on web technologies for more than 20 years. Studying history at Johns Hopkins University, he discovered that he spent more time in the computer lab than at the document archives. Since then, his career has oscillated between programming and training, working with various languages used in web development over the last 15 years. He now concentrates on JavaScript and Java, with the occasional nostalgic visits to Perl and XML. When not exploring a new JavaScript library, writing workbooks or hacking code, he enjoys drinking beer, watching movies, reading books, and traveling the world.

Adam D. Scott is an engineering manager, web developer, and educator based in Connecticut. He currently works as the web development lead at the Consumer Financial Protection Bureau, where he focuses on building open source tools. Additionally, he has worked in education for over a decade, teaching and writing curriculum on a range of technical topics. He is the author of WordPress for Education (Packt, 2012), the Introduction to Modern Front-End Development video course (O’Reilly 2015), the Ethical Web Development report series (O’Reilly, 2016-2017), and JavaScript Everywhere (O’Reilly, 2020).

Shelley Powers has been working with, and writing about, web technologies—from the first release of JavaScript to the latest graphics and design tools—for more than 12 years. Her recent O’Reilly books have covered the semantic web, Ajax, JavaScript, and web graphics. She’s an avid amateur photographer and web development aficionado, who enjoys applying her latest experiments on her many websites.

Colophon

The animal on the cover of *JavaScript Cookbook, Second Edition* is a little egret (*Egretta garzetta*). A small white heron, it is the old world counterpart to the very similar new world snowy egret. It is the smallest and most common egret in Singapore, and its original breeding distribution included the large inland and coastal wetlands in warm temperate parts of Europe, Asia, Africa, Taiwan, and Australia. In warmer locations, most birds are permanent residents; northern populations, including many European birds, migrate to Africa and southern Asia. They may also wander north after the breeding season, which presumably has led to this egret's range expansion.

The adult little egret is 55–65 cm long with an 88–106 cm wingspan. It weighs 350–550 grams. Its plumage is all white. It has long black legs with yellow feet and a slim black bill. In the breeding season, the adult has two long nape plumes and gauzy plumes on the back and breast, and the bare skin between its bill and eyes becomes red or blue. Juvenile egrets are similar to nonbreeding adults but have duller legs and feet. Little egrets are the liveliest hunters among herons and egrets, with a wide variety of techniques: they may patiently stalk prey in shallow waters; stand on one leg and stir the mud with the other to scare up prey; or, better yet, stand on one leg and wave the other bright yellow foot over the water's surface to lure aquatic prey into range. The birds are mostly silent, but make various croaking and bubbling calls at their breeding colonies and produce a harsh alarm call when disturbed.

The little egret nests in colonies, often with other wading birds, usually on platforms of sticks in trees or shrubs, in reed beds, or in bamboo groves. In some locations, such as the Cape Verde Islands, the species nests on cliffs. In pairs they will defend a small breeding territory. Both parents will incubate their 3–5 eggs for 21–25 days until hatching. The eggs are oval in shape and have a pale, nonglossy, blue-green color. The young birds are covered in white down feathers, are cared for by both parents, and fledge after 40 to 45 days. During this stage, the young egret stalks its prey in shallow water, often running with raised wings or shuffling its feet. It may also stand still and wait to ambush prey. It eats fish, insects, amphibians, crustaceans, and reptiles.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from Cassell's *Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.