

Chapter 17. JavaScript Tools and Extensions

Congratulations on reaching the final chapter of this book. If you have read everything that comes before, you now have a detailed understanding of the JavaScript language and know how to use it in Node and in web browsers. This chapter is a kind of graduation present: it introduces a handful of important programming tools that many JavaScript programmers find useful, and also describes two widely used extensions to the core JavaScript language. Whether or not you choose to use these tools and extensions for your own projects, you are almost certain to see them used in other projects, so it is important to at least know what they are.

The tools and language extensions covered in this chapter are:

- ESLint for finding potential bugs and style problems in your code.
- Prettier for formatting your JavaScript code in a standardized way.
- Jest as an all-in-one solution for writing JavaScript unit tests.
- npm for managing and installing the software libraries that your program depends on.
- Code-bundling tools—like webpack, Rollup, and Parcel—that convert your modules of JavaScript code into a single bundle for use on the web.
- Babel for translating JavaScript code that uses brand-new language features (or that uses language extensions) into JavaScript code that can run in current web browsers.
- The JSX language extension (used by the React framework) that allows you to describe user interfaces using JavaScript expressions that look like HTML markup.
- The Flow language extension (or the similar TypeScript extension) that allows you to annotate your JavaScript code with types and check your code for type safety.

This chapter does not document these tools and extensions in any comprehensive way. The goal is simply to explain them in enough depth that you can understand why they are useful and when you might want to use them. Everything covered in this chapter is widely used in the JavaScript programming world, and if you do decide to adopt a tool or extension, you'll find lots of documentation and tutorials online.

17.1 Linting with ESLint

In programming, the term *lint* refers to code that, while technically correct, is unsightly, or a possible bug, or suboptimal in some way. A *linter* is a tool for

detecting lint in your code, and *linting* is the process of running a linter on your code (and then fixing your code to remove the lint so that the linter no longer complains).

The most commonly used linter for JavaScript today is [ESLint](#). If you run it and then take the time to actually fix the issues it points out, it will make your code cleaner and less likely to have bugs. Consider the following code:

```
var x = 'unused';

export function factorial(x) {
    if (x == 1) {
        return 1;
    } else {
        return x * factorial(x-1)
    }
}
```

If you run ESLint on this code, you might get output like this:

```
$ eslint code/ch17/linty.js

code/ch17/linty.js
  1:1  error    Unexpected var, use let or const instead      no-var
  1:5  error    'x' is assigned a value but never used       no-unused-vars
  1:9  warning   Strings must use doublequote               quotes
  4:11 error    Expected '===' and instead saw '=='          eqeqeq
  5:1  error    Expected indentation of 8 spaces but found 6 indent
  7:28 error    Missing semicolon                          semi

✖ 6 problems (5 errors, 1 warning)
  3 errors and 1 warning potentially fixable with the `--fix` option.
```

Linters can seem nitpicky sometimes. Does it really matter whether we used double quotes or single quotes for our strings? On the other hand, getting indentation right is important for readability, and using `==` and `let` instead of `=` and `var` protects you from subtle bugs. And unused variables are dead weight in your code—there is no reason to keep those around.

ESLint defines many linting rules and has an ecosystem of plug-ins that add many more. But ESLint is fully configurable, and you can define a configuration file that tunes ESLint to enforce exactly the rules you want and only those rules.

17.2 JavaScript Formatting with Prettier

One of the reasons that some projects use linters is to enforce a consistent coding style so that when a team of programmers is working on a shared codebase, they use compatible code conventions. This includes code indentation rules, but can also include things like what kind of quotation marks are preferred and whether there should be a space between the `for` keyword and the open parenthesis that follows it.

A modern alternative to enforcing code formatting rules via a linter is to adopt a tool like [Prettier](#) to automatically parse and reformat all of your code.

Suppose you have written the following function, which works, but is formatted unconventionally:

```
function factorial(x)
{
    if(x==1){return 1}
    else{return x*factorial(x-1)}
}
```

Running Prettier on this code fixes the indentation, adds missing semicolons, adds spaces around binary operators and inserts line breaks after { and before }, resulting in much more conventional-looking code:

```
$ prettier factorial.js
function factorial(x) {
  if (x === 1) {
    return 1;
  } else {
    return x * factorial(x - 1);
}
}
```

If you invoke Prettier with the --write option, it will simply reformat the specified file in place rather than printing a reformatted version. If you use git to manage your source code, you can invoke Prettier with the --write option in a commit hook so that code is automatically formatted before being checked in.

Prettier is particularly powerful if you configure your code editor to run it automatically every time you save a file. I find it liberating to write sloppy code and see it fixed automatically for me.

Prettier is configurable, but it only has a few options. You can select the maximum line length, the indentation amount, whether semicolons should be used, whether strings should be single- or double-quoted, and a few other things. In general, Prettier's default options are quite reasonable. The idea is that you just adopt Prettier for your project and then never have to think about code formatting again.

Personally, I really like using Prettier on JavaScript projects. I have not used it for the code in this book, however, because in much of my code I rely on careful hand formatting to align my comments vertically, and Prettier messes them up.

17.3 Unit Testing with Jest

Writing tests is an important part of any nontrivial programming project. Dynamic languages like JavaScript support testing frameworks that dramatically reduce the effort required to write tests, and almost make test writing fun! There are a lot of test tools and libraries for JavaScript, and many are written in a modular way so that it is possible to pick one library as your test runner, another library for assertions, and a third for mocking. In this section, however, we'll describe [Jest](#), which is a popular framework that includes everything you need in a single package.

Suppose you've written the following function:

```
constgetJSON = require("./getJSON.js");

/**
 * getTemperature() takes the name of a city as its input, and returns
 * a Promise that will resolve to the current temperature of that city,
 * in degrees Fahrenheit. It relies on a (fake) web service that returns
 * world temperatures in degrees Celsius.
 */
module.exports = async function getTemperature(city) {
    // Get the temperature in Celsius from the web service
```

```

let c = await getJSON(
  `https://globaltemps.example.com/api/city/${city.toLowerCase()}`)
);
// Convert to Fahrenheit and return that value.
return (c * 5 / 9) + 32; // TODO: double-check this formula
};

```

A good set of tests for this function might verify that `getTemperature()` is fetching the right URL, and that it is converting temperature scales correctly. We can do this with a Jest-based test like the following. This code defines a mock implementation of `getJSON()` so that the test does not actually make a network request. And because `getTemperature()` is an async function, the tests are `async` as well—it can be tricky to test asynchronous functions, but Jest makes it relatively easy:

```

// Import the function we are going to test
const getTemperature = require("./getTemperature.js");

// And mock the getJSON() module that getTemperature() depends on
jest.mock("./getJSON");
const getJSON = require("./getJSON.js");

// Tell the mock getJSON() function to return an already resolved Promise
// with fulfillment value 0.
getJSON.mockResolvedValue(0);

// Our set of tests for getTemperature() begins here
describe("getTemperature()", () => {
  // This is the first test. We're ensuring that getTemperature() calls
  // getJSON() with the URL that we expect
  test("Invokes the correct API", async () => {
    let expectedURL = "https://globaltemps.example.com/api/city/vancouver";
    let t = await(getTemperature("Vancouver"));
    // Jest mocks remember how they were called, and we can check that.
    expect(getJSON).toHaveBeenCalledWith(expectedURL);
  });

  // This second test verifies that getTemperature() converts
  // Celsius to Fahrenheit correctly
  test("Converts C to F correctly", async () => {
    getJSON.mockResolvedValue(0); // If getJSON returns 0C
    expect(await getTemperature("x")).toBe(32); // We expect 32F

    // 100C should convert to 212F
    getJSON.mockResolvedValue(100); // If getJSON returns 100C
    expect(await getTemperature("x")).toBe(212); // We expect 212F
  });
});

```

With the test written, we can use the `jest` command to run it, and we discover that one of our tests fails:

```

$ jest getTemperature
FAIL  ch17/getTemperature.test.js
  getTemperature()
    ✓ Invokes the correct API (4ms)
    ✗ Converts C to F correctly (3ms)

  ● getTemperature() > Converts C to F correctly

    expect(received).toBe(expected) // Object.is equality

      Expected: 212
      Received: 87.55555555555556

      29 |           // 100C should convert to 212F
      30 |           getJSON.mockResolvedValue(100); // If getJSON returns 100C

```

```

> 31 |         expect(await getTemperature("x")).toBe(212); // Expect 212F
     |
32 |     });
33 | });
34 |

      at Object.<anonymous> (ch17/getTemperature.test.js:31:43)

```

```

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
Time:        1.403s
Ran all test suites matching /getTemperature/i.

```

Our `getTemperature()` implementation is using the wrong formula for converting C to F. It multiplies by 5 and divides by 9 rather than multiplying by 9 and dividing by 5. If we fix the code and run Jest again, we can see the tests pass. And, as a bonus, if we add the `--coverage` argument when we invoke `jest`, it will compute and display the code coverage for our tests:

```

$ jest --coverage getTemperature
PASS  ch17/getTemperature.test.js
  getTemperature()
    ✓ Invokes the correct API (3ms)
    ✓ Converts C to F correctly (1ms)

-----|-----|-----|-----|-----|-----|
File    | % Stmt | % Branch | % Funcs | % Lines | Uncovered Line #
-----|-----|-----|-----|-----|-----|
All files | 71.43 | 100 | 33.33 | 83.33 | 
getJSON.js | 33.33 | 100 | 0 | 50 | 2
getTemperature.js | 100 | 100 | 100 | 100 | 

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.508s
Ran all test suites matching /getTemperature/i.

```

Running our test gave us 100% code coverage for the module we were testing, which is exactly what we wanted. It only gave us partial coverage of `getJSON()`, but we mocked that module and were not trying to test it, so that is expected.

17.4 Package Management with npm

In modern software development, it is common for any nontrivial program that you write to depend on third-party software libraries. If you're writing a web server in Node, for example, you might be using the Express framework. And if you're creating a user interface to be displayed in a web browser, you might use a frontend framework like React or LitElement or Angular. A package manager makes it easy to find and install third-party packages like these. Just as importantly, a package manager keeps track of what packages your code depends on and saves this information into a file so that when someone else wants to try your program, they can download your code and your list of dependencies, then use their own package manager to install all the third-party packages that your code needs.

npm is the package manager that is bundled with Node, and was introduced in §16.1.5. It is just as useful for client-side JavaScript programming as it is for server-side programming with Node, however.

If you are trying out someone else's JavaScript project, then one of the first things you will often do after downloading their code is to type `npm install`. This reads the dependencies listed in the `package.json` file and downloads the third-party packages that the project needs and saves them in a `node_modules/` directory.

You can also type `npm install <package-name>` to install a particular package to your project's `node_modules/` directory:

```
$ npm install express
```

In addition to installing the named package, npm also makes a record of the dependency in the `package.json` file for the project. Recording dependencies in this way is what allows others to install those dependencies simply by typing `npm install`.

The other kind of dependency is on developer tools that are needed by developers who want to work on your project, but aren't actually needed to run the code. If a project uses Prettier, for example, to ensure that all of its code is consistently formatted, then Prettier is a "dev dependency," and you can install and record one of these with `--save-dev`:

```
$ npm install --save-dev prettier
```

Sometimes you might want to install developer tools globally so that they are accessible anywhere even for code that is not part of a formal project with a `package.json` file and a `node_modules/` directory. For that you can use the `-g` (for global) option:

```
$ npm install -g eslint jest
/usr/local/bin/eslint -> /usr/local/lib/node_modules/eslint/bin/eslint.js
/usr/local/bin/jest -> /usr/local/lib/node_modules/jest/bin/jest.js
+ jest@24.9.0
+ eslint@6.7.2
added 653 packages from 414 contributors in 25.596s

$ which eslint
/usr/local/bin/eslint
$ which jest
/usr/local/bin/jest
```

In addition to the "install" command, npm supports "uninstall" and "update" commands, which do what their names say. npm also has an interesting "audit" command that you can use to find and fix security vulnerabilities in your dependencies:

```
$ npm audit --fix
=====
found 0 vulnerabilities
in 876354 scanned packages
```

When you install a tool like ESLint locally for a project, the `eslint` script winds up in `./node_modules/.bin/eslint`, which makes the command awkward to run. Fortunately, npm is bundled with a command known as "npx," which you can use to run locally installed tools with commands like `npx eslint` or `npx jest`. (And if you use npx to invoke a tool that has not been installed yet, it will install it for you.)

The company behind npm also maintains the <https://npmjs.com> package repository, which holds hundreds of thousands of open source packages. But you

don't have to use the npm package manager to access this repository of packages. Alternatives include [yarn](#) and [pnpm](#).

17.5 Code Bundling

If you are writing a large JavaScript program to run in web browsers, you will probably want to use a code-bundling tool, especially if you use external libraries that are delivered as modules. Web developers have been using ES6 modules ([§10.3](#)) for years, since well before the `import` and `export` keywords were supported on the web. In order to do this, programmers use a code-bundler tool that starts at the main entry point (or entry points) of the program and follows the tree of `import` directives to find all modules that the program depends on. It then combines all of those individual module files into a single bundle of JavaScript code and rewrites the `import` and `export` directives to make the code work in this new form. The result is a single file of code that can be loaded into a web browser that does not support modules.

ES6 modules are nearly universally supported by web browsers today, but web developers still tend to use code bundlers, at least when releasing production code. Developers find that user experience is best when a single medium-sized bundle of code is loaded when a user first visits a website than when many small modules are loaded.

Note

Web performance is a notoriously tricky topic and there are lots of variables to consider, including ongoing improvements by browser vendors, so the only way to be sure of the fastest way to load your code is by testing thoroughly and measuring carefully. Keep in mind that there is one variable that is completely under your control: code size. Less JavaScript code will always load and run faster than more JavaScript code!

There are a number of good JavaScript bundler tools available. Commonly used bundlers include [webpack](#), [Rollup](#) and [Parcel](#). The basic features of bundlers are more or less the same, and they are differentiated based on how configurable they are or how easy they are to use. Webpack has been around for a long time, has a large ecosystem of plug-ins, is highly configurable, and can support older nonmodule libraries. But it can also be complex and hard to configure. At the other end of the spectrum is Parcel which is intended as a zero-configuration alternative that simply does the right thing.

In addition to performing basic bundling, bundler tools can also provide some additional features:

- Some programs have more than one entry point. A web application with multiple pages, for example, could be written with a different entry point for each page. Bundlers generally allow you to create one bundle per entry point or to create a single bundle that supports multiple entry points.
- Programs can use `import()` in its functional form ([§10.3.6](#)) instead of its static form to dynamically load modules when they are actually needed rather than statically loading them at program startup time. Doing this is often a good way to improve the startup time for your program. Bundler tools that support `import()` may be able to produce multiple output bundles: one to load at startup time, and one or more that are loaded dynamically when needed. This can work well if there are only a few calls to `import()` in

your program and they load modules with relatively disjoint sets of dependencies. If the dynamically loaded modules share dependencies then it becomes tricky to figure out how many bundles to produce, and you are likely to have to manually configure your bundler to sort this out.

- Bundlers can generally output a *source map* file that defines a mapping between the lines of code in the bundle and the corresponding lines in the original source files. This allows browser developer tools to automatically display JavaScript errors at their original unbundled locations.
- Sometimes when you import a module into your program, you only use a few of its features. A good bundler tool can analyze the code to determine which parts are unused and can be omitted from the bundles. This feature goes by the whimsical name of “tree-shaking.”
- Bundlers typically have a plug-in-based architecture and support plug-ins that allow importing and bundling “modules” that are not actually files of JavaScript code. Suppose that your program includes a large JSON-compatible data structure. Code bundlers can be configured to allow you to move that data structure into a separate JSON file and then import it into your program with a declaration like `import widgets from "./big-widget-list.json"`. Similarly, web developers who embed CSS into their JavaScript programs can use bundler plug-ins that allow them to import CSS files with an `import` directive. Note, however, that if you import anything other than a JavaScript file, you are using a nonstandard JavaScript extension and making your code dependent on the bundler tool.
- In a language like JavaScript that does not require compilation, running a bundler tool feels like a compilation step, and it is frustrating to have to run a bundler after every code edit before you can run the code in your browser. Bundlers typically support filesystem watchers that detect edits to any files in a project directory and automatically regenerate the necessary bundles. With this feature in place you can typically save your code and then immediately reload your web browser window to try it out.
- Some bundlers also support a “hot module replacement” mode for developers where each time a bundle is regenerated, it is automatically loaded into the browser. When this works, it is a magical experience for developers, but there are some tricks going on under the hood to make it work, and it is not suitable for all projects.

17.6 Transpilation with Babel

[Babel](#) is a tool that compiles JavaScript written using modern language features into JavaScript that does not use those modern language features. Because it compiles JavaScript to JavaScript, Babel is sometimes called a “transpiler.” Babel was created so that web developers could use the new language features of ES6 and later while still targeting web browsers that only supported ES5.

Language features such as the `**` exponentiation operator and arrow functions can be transformed relatively easily into `Math.pow()` and `function` expressions. Other language features, such as the `class` keyword, require much more complex transformations, and, in general, the code output by Babel is not meant to be human readable. Like bundler tools, however, Babel can produce source maps that map transformed code locations back to their original source locations, and this helps dramatically when working with transformed code.

Browser vendors are doing a better job of keeping up with the evolution of the JavaScript language, and there is much less need today to compile away arrow functions and class declarations. Babel can still help when you want to use the very latest features like underscore separators in numeric literals.

Like most of the other tools described in this chapter, you can install Babel with npm and run it with npx. Babel reads a `.babelrc` configuration file that tells it how you would like your JavaScript code transformed. Babel defines “presets” that you can choose from depending on which language extensions you want to use and how aggressively you want to transform standard language features. One of Babel’s interesting presets is for code compression by minification (stripping comments and whitespace, renaming variables, and so on).

If you use Babel and a code-bundling tool, you may be able to set up the code bundler to automatically run Babel on your JavaScript files as it builds the bundle for you. If so, this can be a convenient option because it simplifies the process of producing runnable code. Webpack, for example, supports a “babel-loader” module that you can install and configure to run Babel on each JavaScript module as it is bundled up.

Even though there is less need to transform the core JavaScript language today, Babel is still commonly used to support nonstandard extensions to the language, and we’ll describe two of these language extensions in the sections that follow.

17.7 JSX: Markup Expressions in JavaScript

JSX is an extension to core JavaScript that uses HTML-style syntax to define a tree of elements. JSX is most closely associated with the React framework for user interfaces on the web. In React, the trees of elements defined with JSX are ultimately rendered into a web browser as HTML. Even if you have no plans to use React yourself, its popularity means that you are likely to see code that uses JSX. This section explains what you need to know to make sense of it. (This section is about the JSX language extension, not about React, and it explains only enough of React to provide context for the JSX syntax.)

You can think of a JSX element as a new type of JavaScript expression syntax. JavaScript string literals are delimited with quotation marks, and regular expression literals are delimited with slashes. In the same way, JSX expression literals are delimited with angle brackets. Here is a very simple one:

```
let line = <hr/>;
```

If you use JSX, you will need to use Babel (or a similar tool) to compile JSX expressions into regular JavaScript. The transformation is simple enough that some developers choose to use React without using JSX. Babel transforms the JSX expression in this assignment statement into a simple function call:

```
let line = React.createElement("hr", null);
```

JSX syntax is HTML-like, and like HTML elements, React elements can have attributes like these:

```
let image = ;
```

When an element has one or more attributes, they become properties of an object passed as the second argument to `createElement()`:

```
let image = React.createElement("img", {
  src: "logo.png",
  alt: "The JSX logo",
  hidden: true
});
```

Like HTML elements, JSX elements can have strings and other elements as children. Just as JavaScript's arithmetic operators can be used to write arithmetic expressions of arbitrary complexity, JSX elements can also be nested arbitrarily deeply to create trees of elements:

```
let sidebar = (
  <div className="sidebar">
    <h1>Title</h1>
    <hr/>
    <p>This is the sidebar content</p>
  </div>
);
```

Regular JavaScript function call expressions can also be nested arbitrarily deeply, and these nested JSX expressions translate into a set of nested `createElement()` calls. When an JSX element has children, those children (which are typically strings and other JSX elements) are passed as the third and subsequent arguments:

```
let sidebar = React.createElement(
  "div", { className: "sidebar"}, // This outer call creates a <div>
  React.createElement("h1", null, // This is the first child of the <div/>
    "Title"), // and its own first child.
  React.createElement("hr", null), // The second child of the <div/>.
  React.createElement("p", null, // And the third child.
    "This is the sidebar content"));
```

The value returned by `React.createElement()` is an ordinary JavaScript object that is used by React to render output in a browser window. Since this section is about the JSX syntax and not about React, we're not going to go into any detail about the returned Element objects or the rendering process. It is worth noting that you can configure Babel to compile JSX elements to invocations of a different function, so if you think that JSX syntax would be a useful way to express other kinds of nested data structures, you can adopt it for your own non-React uses.

An important feature of JSX syntax is that you can embed regular JavaScript expressions within JSX expressions. Within a JSX expression, text within curly braces is interpreted as plain JavaScript. These nested expressions are allowed as attribute values and as child elements. For example:

```
function sidebar(className, title, content, drawLine=true) {
  return (
    <div className={className}>
      <h1>{title}</h1>
      { drawLine && <hr/> }
      <p>{content}</p>
    </div>
  );
}
```

The `sidebar()` function returns a JSX element. It takes four arguments that it uses within the JSX element. The curly brace syntax may remind you of template literals that use `${}` to include JavaScript expressions within strings. Since we know that JSX expressions compile into function invocations, it should not be surprising that arbitrary JavaScript expressions can be included because function invocations can be written with arbitrary expressions as well. This example code is translated by Babel into the following:

```
function sidebar(className, title, content, drawLine=true) {
  return React.createElement("div", { className: className },
    React.createElement("h1", null, title),
    drawLine && React.createElement("hr", null),
    React.createElement("p", null, content));
}
```

This code is easy to read and understand: the curly braces are gone and the resulting code passes the incoming function parameters to `React.createElement()` in a natural way. Note the neat trick that we've done here with the `drawLine` parameter and the short-circuiting `&&` operator. If you call `sidebar()` with only three arguments, then `drawLine` defaults to `true`, and the fourth argument to the outer `createElement()` call is the `<hr/>` element. But if you pass `false` as the fourth argument to `sidebar()`, then the fourth argument to the outer `createElement()` call evaluates to `false`, and no `<hr/>` element is ever created. This use of the `&&` operator is a common idiom in JSX to conditionally include or exclude a child element depending on the value of some other expression. (This idiom works with React because React simply ignores children that are `false` or `null` and does not produce any output for them.)

When you use JavaScript expressions within JSX expressions, you are not limited to simple values like the string and boolean values in the preceding example. Any JavaScript value is allowed. In fact, it is quite common in React programming to use objects, arrays, and functions. Consider the following function, for example:

```
// Given an array of strings and a callback function return a JSX element
// representing an HTML <ul> list with an array of <li> elements as its child.
function list(items, callback) {
  return (
    <ul style={ {padding:10, border:"solid red 4px"} }>
      {items.map((item,index) => {
        <li onClick={() => callback(index)} key={index}>{item}</li>
      })}
    </ul>
  );
}
```

This function uses an object literal as the value of the `style` attribute on the `` element. (Note that double curly braces are required here.) The `` element has a single child, but the value of that child is an array. The child array is the array created by using the `map()` function on the input array to create an array of `` elements. (This works with React because the React library flattens the children of an element when it renders them. An element with one array child is the same as that element with each of those array elements as children.) Finally, note that each of the nested `` elements has an `onClick` event handler attribute whose value is an arrow function. The JSX code compiles to the following pure JavaScript code (which I have formatted with Prettier):

```
function list(items, callback) {
  return React.createElement(
    "ul",
    { style: { padding: 10, border: "solid red 4px" } },
    items.map((item, index) =>
      React.createElement(
        "li",
        { onClick: () => callback(index), key: index },
        item
      )
    )
  );
}
```

One other use of object expressions in JSX is with the object spread operator ([§6.10.4](#)) to specify multiple attributes at once. Suppose that you find yourself writing a lot of JSX expressions that repeat a common set of attributes. You can simplify your expressions by defining the attributes as properties of an object and “spreading them into” your JSX elements:

```
let hebrew = { lang: "he", dir: "rtl" }; // Specify language and direction
let shalom = <span className="emphasis" {...hebrew}>שלום</span>;
```

Babel compiles this to use an `_extends()` function (omitted here) that combines that `className` attribute with the attributes contained in the `hebrew` object:

```
let shalom = React.createElement("span",
  _extends({className: "emphasis"}, hebrew),
  "\u05E9\u05DC\u05D5\u05DD");
```

Finally, there is one more important feature of JSX that we have not covered yet. As you’ve seen, all JSX elements begin with an identifier immediately after the opening angle bracket. If the first letter of this identifier is lowercase (as it has been in all of the examples here), then the identifier is passed to `createElement()` as a string. But if the first letter of the identifier is uppercase, then it is treated as an actual identifier, and it is the JavaScript value of that identifier that is passed as the first argument to `createElement()`. This means that the JSX expression `<Math/>` compiles to JavaScript code that passes the global `Math` object to `React.createElement()`.

For React, this ability to pass non-string values as the first argument to `createElement()` enables the creation of *components*. A component is a way of writing a simple JSX expression (with an uppercase component name) that represents a more complex expression (using lowercase HTML tag names).

The simplest way to define a new component in React is to write a function that takes a “props object” as its argument and returns a JSX expression. A *props object* is simply a JavaScript object that represents attribute values, like the objects that are passed as the second argument to `createElement()`. Here, for example, is another take on our `sidebar()` function:

```
function Sidebar(props) {
  return (
    <div>
      <h1>{props.title}</h1>
      { props.drawLine && <hr/> }
      <p>{props.content}</p>
    </div>
  );
}
```

This new `Sidebar()` function is a lot like the earlier `sidebar()` function. But this one has a name that begins with a capital letter and takes a single object argument instead of separate arguments. This makes it a React component and means that it can be used in place of an HTML tag name in JSX expressions:

```
let sidebar = <Sidebar title="Something snappy" content="Something wise"/>;
```

This `<Sidebar/>` element compiles like this:

```
let sidebar = React.createElement(Sidebar, {
  title: "Something snappy",
  content: "Something wise"
});
```

It is a simple JSX expression, but when React renders it, it will pass the second argument (the `Props` object) to the first argument (the `Sidebar()` function) and will use the JSX expression returned by that function in place of the `<Sidebar>` expression.

17.8 Type Checking with Flow

[Flow](#) is a language extension that allows you to annotate your JavaScript code with type information, and a tool for checking your JavaScript code (both annotated and unannotated) for type errors. To use Flow, you start writing code using the Flow language extension to add type annotations. Then you run the Flow tool to analyze your code and report type errors. Once you have fixed the errors and are ready to run the code, you use Babel (perhaps automatically as part of the code-bundling process) to strip the Flow type annotations out of your code. (One of the nice things about the Flow language extension is that there isn't any new syntax that Flow has to compile or transform. You use the Flow language extension to add annotations to the code, and all Babel has to do is to strip those annotations out to return your code to standard JavaScript.)

TypeScript Versus Flow

TypeScript is a very popular alternative to Flow. TypeScript is an extension of JavaScript that adds types as well as other language features. The TypeScript compiler “tsc” compiles TypeScript programs into JavaScript programs and in the process analyzes them and reports type errors in much the same the way that Flow does. tsc is not a Babel plugin: it is its own standalone compiler.

Simple type annotations in TypeScript are usually written identically to the same annotations in Flow. For more advanced typing, the syntax of the two extensions diverges, but the intent and value of the two extensions is the same. My goal in this section is to explain the benefits of type annotations and static code analysis. I'll be doing that with examples based on Flow, but everything demonstrated here can also be achieved with TypeScript with relatively simple syntax changes.

TypeScript was released in 2012, before ES6, when JavaScript did not have a `class` keyword or a `for/of` loop or modules or Promises. Flow is a narrow language extension that adds type annotations to JavaScript and nothing else. TypeScript, by contrast, was very much designed as a new language. As its name implies, adding types to JavaScript is the primary purpose of TypeScript, and it is the reason that people use it today. But types are not the only feature that TypeScript adds to JavaScript: the TypeScript language has `enum` and `namespace` keywords that simply do not exist in JavaScript. In 2020, TypeScript has better integration with IDEs and code editors (particularly VSCode, which, like TypeScript, is from Microsoft) than Flow does.

Ultimately, this is a book about JavaScript, and I'm covering Flow here instead of TypeScript because I don't want to take the focus off of JavaScript. But everything you learn here about adding types to JavaScript will be helpful to you if you decide to adopt TypeScript for your projects.

Using Flow requires commitment, but I have found that for medium and large projects, the extra effort is worth it. It takes extra time to add type annotations to your code, to run Flow every time you edit the code, and to fix the type errors it reports. But in return Flow will enforce good coding discipline and will not allow you to cut corners that can lead to bugs. When I have worked on projects that use Flow, I have been impressed by the number of errors it found in my own code.

Being able to fix those issues before they became bugs is a great feeling and gives me extra confidence that my code is correct.

When I first started using Flow, I found that it was sometimes difficult to understand why it was complaining about my code. With some practice, though, I came to understand its error messages and found that it was usually easy to make minor changes to my code to make it safer and to satisfy Flow.¹ I do not recommend using Flow if you still feel like you are learning JavaScript itself. But once you are confident with the language, adding Flow to your JavaScript projects will push you to take your programming skills to the next level. And this, really, is why I'm dedicating the last section of this book to a Flow tutorial: because learning about JavaScript type systems offers a glimpse of another level, or another style, of programming.

This section is a tutorial, and it does not attempt to cover Flow comprehensively. If you decide to try Flow, you will almost certainly end up spending time reading the documentation at <https://flow.org>. On the other hand, you do not need to master the Flow type system before you can start making practical use of it in your projects: the simple uses of Flow described here will take you a long way.

17.8.1 Installing and Running Flow

Like the other tools described in this chapter, you can install the Flow type-checking tool using a package manager, with a command like `npm install -g flow-bin` or `npm install --save-dev flow-bin`. If you install the tool globally with `-g`, then you can run it with `flow`. And if you install it locally in your project with `--save-dev`, then you can run it with `npx flow`. Before using Flow to do type checking, the first time run it as `flow --init` in the root directory of your project to create a `.flowconfig` configuration file. You may never need to add anything to this file, but Flow needs it to know where your project root is.

When you run Flow, it will find all the JavaScript source code in your project, but it will only report type errors for the files that have “opted in” to type checking by adding a `// @flow` comment at the top of the file. This opt-in behavior is important because it means that you can adopt Flow for existing projects and then begin to convert your code one file at a time, without being bothered by errors and warnings on files that have not yet been converted.

Flow may be able to find errors in your code even if all you do is opt in with a `// @flow` comment. Even if you do not use the Flow language extension and add no type annotations to your code, the Flow type checker tool can still make inferences about the values in your program and alert you when you use them inconsistently.

Consider the following Flow error message:

```
Error ..... variableReassignment.js:6:3
Cannot assign 1 to i.r because:
  • property r is missing in number [1].
2| let i = { r: 0, i: 1 };    // The complex number 0+1i
[1] 3| for(i = 0; i < 10; i++) { // Oops! The loop variable overwrites i
  4|   console.log(i);
  5|
  6| i.r = 1;                  // Flow detects the error here
```

In this case, we declare the variable `i` and assign an object to it. Then we use `i` again as a loop variable, overwriting the object. Flow notices this and flags an

error when we try to use `i` as if it still held an object. (A simple fix would be to write `for(let i = 0;` making the loop variable local to the loop.)

Here is another error that Flow detects even without type annotations:

```
Error ..... size.js:3:14
```

Cannot get `x.length` because property `length` is missing in `Number` [1].

```
1 // @flow
2 function size(x) {
3     return x.length;
4 }
[1] 5 let s = size(1000);
```

Flow sees that the `size()` function takes a single argument. It doesn't know the type of that argument, but it can see that the argument is expected to have a `length` property. When it sees this `size()` function being called with a numeric argument, it correctly flags this as an error because numbers do not have `length` properties.

17.8.2 Using Type Annotations

When you declare a JavaScript variable, you can add a Flow type annotation to it by following the variable name with a colon and the type:

```
let message: string = "Hello world";
let flag: boolean = false;
let n: number = 42;
```

Flow would know the types of these variables even if you did not annotate them: it can see what values you assign to each variable, and it keeps track of that. If you add type annotations, however, Flow knows both the type of the variable and that you have expressed the intent that the variable should always be of that type. So if you use the type annotation, Flow will flag an error if you ever assign a value of a different type to that variable. Type annotations for variables are also particularly useful if you tend to declare all your variables up at the top of a function before they are used.

Type annotations for function arguments are like annotations for variables: follow the name of the function argument with a colon and the type name. When annotating a function, you typically also add an annotation for the return type of the function. This goes between the close parenthesis and the open curly brace of the function body. Functions that return nothing use the Flow type `void`.

In the preceding example we defined a `size()` function that expected an argument with a `length` property. Here's how we could change that function to explicitly specify that it expects a string argument and returns a number. Note, Flow now flags an error if we pass an array to the function, even though the function would work in that case:

```
Error ..... size2.js:5:18
```

Cannot call `size` with array literal bound to `s` because array literal [1] is incompatible with string [2].

```
[2] 2 function size(s: string): number {
3     return s.length;
4 }
[1] 5 console.log(size([1,2,3]));
```

Using type annotations with arrow functions is also possible, though it can turn this normally succinct syntax into something more verbose:

```
const size = (s: string): number => s.length;
```

An important thing to understand about Flow is that the JavaScript value `null` has the Flow type `null` and the JavaScript value `undefined` has the Flow type `void`. But neither of these values is a member of any other type (unless you explicitly add it). If you declare a function parameter to be a string, then it must be a string, and it is an error to pass `null` or to pass `undefined` or to omit the argument (which is basically the same thing as passing `undefined`):

Error size3.js:3:18

Cannot call size with null bound to s because null [1] is incompatible with string [2].

```
1 // @flow
[2] 2 const size = (s: string): number => s.length;
[1] 3 console.log(size(null));
```

If you want to allow `null` and `undefined` as legal values for a variable or function argument, simply prefix the type with a question mark. For example, use `?string` or `?number` instead of `string` or `number`. If we change our `size()` function to expect an argument of type `?string`, then Flow doesn't complain when we pass `null` to the function. But it now has something else to complain about:

Error size4.js:3:14

Cannot get s.length because property length is missing in null or undefined [1].

```
1 // @flow
[1] 2 function size(s: ?string): number {
  3   return s.length;
  4 }
[5] 5 console.log(size(null));
```

What Flow is telling us here is that it is not safe to write `s.length` because, at this place in our code, `s` might be `null` or `undefined`, and those values do not have `length` properties. This is where Flow makes sure we do not cut any corners. If a value might be `null`, Flow will insist that we check for that case before we do anything that depends on the value not being `null`.

In this case, we can fix the issue by changing the body of the function as follows:

```
function size(s: ?string): number {
  // At this point in the code, s could be a string or null or undefined.
  if (s === null || s === undefined) {
    // In this block, Flow knows that s is null or undefined.
    return -1;
  } else {
    // And in this block, Flow knows that s is a string.
    return s.length;
  }
}
```

When the function is first called, the parameter can have more than one type. But by adding type-checking code, we create a block within the code where Flow knows for sure that the parameter is a string. When we use `s.length` within that block, Flow does not complain. Note that Flow does not require you to write verbose code like this. Flow would also be satisfied if we just replaced the body of the `size()` function with `return s ? s.length : -1;`.

Flow syntax allows a question mark before any type specification to indicate that, in addition to the specified type, `null` and `undefined` are allowed as well. Question marks can also appear after a parameter name to indicate that the parameter itself

is optional. So if we changed the declaration of the parameter `s` from `s: ?string` to `s? : string`, that would mean it is OK to call `size()` with no arguments (or with the value `undefined`, which is the same as omitting it), but that if we do call it with a parameter other than `undefined`, then that parameter must be a string. In this case, `null` is not a legal value.

So far, we've discussed primitive types `string`, `number`, `boolean`, `null`, and `void` and have demonstrated how you can use them with variable declarations, function parameters, and function return values. The subsections that follow describe some more complex types supported by Flow.

17.8.3 Class Types

In addition to the primitive types that Flow knows about, it also knows about all of JavaScript's built-in classes and allows you to use class names as types. The following function, for example, uses type annotations to indicate that it should be invoked with one `Date` object and one `RegExp` object:

```
// @flow
// Return true if the ISO representation of the specified date
// matches the specified pattern, or false otherwise.
// E.g: const isTodayChristmas = dateMatches(new Date(), /^\\d{4}-12-25T/);
export function dateMatches(d: Date, p: RegExp): boolean {
    return p.test(d.toISOString());
}
```

If you define your own classes with the `class` keyword, those classes automatically become valid Flow types. In order to make this work, however, Flow does require you to use type annotations in the class. In particular, each property of the class must have its type declared. Here is a simple complex number class that demonstrates this:

```
// @flow
export default class Complex {
    // Flow requires an extended class syntax that includes type annotations
    // for each of the properties used by the class.
    i: number;
    r: number;
    static i: Complex;

    constructor(r: number, i:number) {
        // Any properties initialized by the constructor must have Flow type
        // annotations above.
        this.r = r;
        this.i = i;
    }

    add(that: Complex) {
        return new Complex(this.r + that.r, this.i + that.i);
    }
}

// This assignment would not be allowed by Flow if there was not a
// type annotation for i inside the class.
Complex.i = new Complex(0,1);
```

17.8.4 Object Types

The Flow type to describe an object looks a lot like an object literal, except that property values are replaced by property types. Here, for example, is a function that expects an object with numeric `x` and `y` properties:

```
// @flow
// Given an object with numeric x and y properties, return the
// distance from the origin to the point (x,y) as a number.
export default function distance(point: {x:number, y:number}): number {
    return Math.hypot(point.x, point.y);
}
```

In this code, the text `{x:number, y:number}` is a Flow type, just like `string` or `Date` is. As with any type, you can add a question mark at the front to indicate that `null` and `undefined` should also be allowed.

Within an object type, you can follow any of the property names with a question mark to indicate that that property is optional and may be omitted. For example, you might write the type for an object that represents a 2D or 3D point like this:

```
{x: number, y: number, z?: number}
```

If a property is not marked as optional in an object type, then it is required, and Flow will report an error if an appropriate property is not present in the actual value. Normally, however, Flow tolerates extra properties. If you were to pass an object that had a `w` property to the `distance()` function above, Flow would not complain.

If you want Flow to strictly enforce that an object does not have properties other than those explicitly declared in its type, you can declare an *exact object type* by adding vertical bars to the curly braces:

```
{| x: number, y: number |}
```

JavaScript's objects are sometimes used as dictionaries or string-to-value maps. When used like this, the property names are not known in advance and cannot be declared in a Flow type. If you use objects this way, you can still use Flow to describe the data structure. Suppose that you have an object where the properties are the names of the world's major cities and the values of those properties are objects that specify the geographical location of those cities. You might declare this data structure like this:

```
// @flow
const cityLocations : {[string]: {longitude:number, latitude:number}} = {
    "Seattle": { longitude: 47.6062, latitude: -122.3321 },
    // TODO: if there are any other important cities, add them here.
};
export default cityLocations;
```

17.8.5 Type Aliases

Objects can have many properties, and the Flow type that describes such an object will be long and difficult to type. And even relatively short object types can be confusing because they look so much like object literals. Once we get beyond simple types like `number` and `?string`, it is often useful to be able to define names for our Flow types. And in fact, Flow uses the `type` keyword to do exactly that. Follow the `type` keyword with an identifier, an equals sign, and a Flow type. Once you've done that, the identifier will be an alias for the type. Here, for example, is how we could rewrite the `distance()` function from the previous section with an explicitly defined `Point` type:

```
// @flow
export type Point = {
    x: number,
    y: number
};
```

```
// Given a Point object return its distance from the origin
export default function distance(point: Point): number {
    return Math.hypot(point.x, point.y);
}
```

Note that this code exports the `distance()` function and also exports the `Point` type. Other modules can use `import type Point from './distance.js'` if they want to use that type definition. Keep in mind, though, that `import type` is a Flow language extension and not a real JavaScript import directive. Type imports and exports are used by the Flow type checker, but like all other Flow language extensions, they are stripped out of the code before it ever runs.

Finally, it is worth noting that instead of defining a name for a Flow object type that represents a point, it would probably be simpler and cleaner to just define a `Point` class and use that class as the type.

17.8.6 Array Types

The Flow type to describe an array is a compound type that also includes the type of the array elements. Here, for example, is a function that expects an array of numbers, and the error that Flow reports if you try to call the function with an array that has non-numeric elements:

```
Error ..... average.js:8:16
```

Cannot call average with array literal bound to data because string [1] is incompatible with number [2] in array element.

```
[2] 2 | function average(data: Array<number>) {
[3] 3 |     let sum = 0;
[4] 4 |     for(let x of data) sum += x;
[5] 5 |     return sum/data.length;
[6] 6 |
[7] 7 |
[1] 8 | average([1, 2, "three"]);
```

The Flow type for an array is `Array` followed by the element type in angle brackets. You can also express an array type by following the element type with open and close square brackets. So in this example we could have written `number[]` instead of `Array<number>`. I prefer the angle bracket notation because, as we'll see, there are other Flow types that use this angle-bracket syntax.

The `Array` type syntax shown works for arrays with an arbitrary number of elements, all of which have the same type. Flow has a different syntax for describing the type of a *tuple*: an array with a fixed number of elements, each of which may have a different type. To express the type of a tuple, simply write the type of each of its elements, separate them with commas, and enclose them all in square brackets.

A function that returns an HTTP status code and message might look like this, for example:

```
function getStatus():[number, string] {
    return [getStatusCode(), getStatusMessage()];
}
```

Functions that return tuples are awkward to work with unless you use destructuring assignment:

```
let [code, message] = getStatus();
```

Destructuring assignment, plus Flow's type-aliasing capabilities, make tuples easy enough to work with that you might consider them as an alternative to classes for simple datatypes:

```
// @flow
export type Color = [number, number, number, number]; // [r, g, b, opacity]

function gray(level: number): Color {
    return [level, level, level, 1];
}

function fade([r,g,b,a]: Color, factor: number): Color {
    return [r, g, b, a/factor];
}

let [r, g, b, a] = fade(gray(75), 3);
```

Now that we have a way to express the type of an array, let's return to the `size()` function from earlier and modify it to expect an array argument instead of a string argument. We want the function to be able to accept an array of any length, so a tuple type is not appropriate. But we don't want to restrict our function to working only for arrays where all the elements have the same type. The solution is the type `Array<mixed>`:

```
// @flow
function size(s: Array<mixed>): number {
    return s.length;
}
console.log(size([1,true,"three"]));
```

The element type `mixed` indicates that the elements of the array can be of any type. If our function actually indexed the array and attempted to use any of those elements, Flow would insist that we use `typeof` checks or other tests to determine the type of the element before performing any unsafe operation on it. (If you are willing to give up on type checking, you can also use `any` instead of `mixed`: it allows you to do whatever you want with the values of the array without ensuring that the values are of the type you expect.)

17.8.7 Other Parameterized Types

We've seen that when you annotate a value as an `Array`, Flow requires you to also specify the type of the array elements inside angle brackets. This additional type is known as a *type parameter*, and `Array` is not the only JavaScript class that is parameterized.

JavaScript's `Set` class is a collection of elements, like an array is, and you can't use `Set` as a type by itself, but you have to include a type parameter within angle brackets to specify the type of the values contained in the set. (Though you can use `mixed` or `any` if the set may contain values of multiple types.) Here's an example:

```
// @flow
// Return a set of numbers with members that are exactly twice those
// of the input set of numbers.
function double(s: Set<number>): Set<number> {
    let doubled: Set<number> = new Set();
    for(let n of s) doubled.add(n * 2);
    return doubled;
}
console.log(double(new Set([1,2,3]))); // Prints "Set {2, 4, 6}"
```

Map is another parameterized type. In this case, there are two type parameters that must be specified; the type of the keys and the types of the values:

```
// @flow
import type { Color } from "./Color.js";

let colorNames: Map<string, Color> = new Map([
    ["red", [1, 0, 0, 1]],
    ["green", [0, 1, 0, 1]],
    ["blue", [0, 0, 1, 1]]
]);
```

Flow lets you define type parameters for your own classes as well. The following code defines a Result class but parameterizes that class with an Error type and a Value type. We use placeholders E and V in the code to represent these type parameters. When the user of this class declares a variable of type Result, they will specify the actual types to substitute for E and V. The variable declaration might look like this:

```
let result: Result<TypeError, Set<string>>;
```

And here is how the parameterized class is defined:

```
// @flow
// This class represents the result of an operation that can either
// throw an error of type E or a value of type V.
export class Result<E, V> {
    error: ?E;
    value: ?V;

    constructor(error: ?E, value: ?V) {
        this.error = error;
        this.value = value;
    }

    threw(): ?E { return this.error; }
    returned(): ?V { return this.value; }

    get(): V {
        if (this.error) {
            throw this.error;
        } else if (this.value === null || this.value === undefined) {
            throw new TypeError("Error and value must not both be null");
        } else {
            return this.value;
        }
    }
}
```

And you can even define type parameters for functions:

```
// @flow
// Combine the elements of two arrays into an array of pairs
function zip<A,B>(a:Array<A>, b:Array<B>): Array<[?A,?B]> {
    let result: Array<[?A,?B]> = [];
    let len = Math.max(a.length, b.length);
    for(let i = 0; i < len; i++) {
        result.push([a[i], b[i]]);
    }
    return result;
}

// Create the array [[1,'a'], [2,'b'], [3,'c'], [4,undefined]]
let pairs: Array<[?number,?string]> = zip([1,2,3,4], ['a','b','c'])
```

17.8.8 Read-Only Types

Flow defines some special parameterized “utility types” that have names beginning with `$`. Most of these types have advanced use cases that we are not going to cover here. But two of them are quite useful in practice. If you have an object type `T` and want to make a read-only version of that type, just write `$ReadOnly<T>`. Similarly, you can write `$ReadOnlyArray<T>` to describe a read-only array with elements of type `T`.

The reason to use these types is not because they can offer any guarantee that an object or array can’t be modified (see `Object.freeze()` in [§14.2](#) if you want true read-only objects) but because it allows you to catch bugs caused by unintentional modifications. If you write a function that takes an object or array argument and does not change any of the object’s properties or the array’s elements, then you can annotate the function parameter with one of Flow’s read-only types. If you do this, then Flow will report an error if you forget and accidentally modify the input value. Here are two examples:

```
// @flow
type Point = {x:number, y:number};

// This function takes a Point object but promises not to modify it
function distance(p: $ReadOnly<Point>): number {
    return Math.hypot(p.x, p.y);
}

let p: Point = {x:3, y:4};
distance(p) // => 5

// This function takes an array of numbers that it will not modify
function average(data: $ReadOnlyArray<number>): number {
    let sum = 0;
    for(let i = 0; i < data.length; i++) sum += data[i];
    return sum/data.length;
}

let data: Array<number> = [1,2,3,4,5];
average(data) // => 3
```

17.8.9 Function Types

We have seen how to add type annotations to specify the types of a function’s parameters and its return type. But when one of the parameters of a function is itself a function, we need to be able to specify the type of that function parameter.

To express the type of a function with Flow, write the types of each parameter, separate them with commas, enclose them in parentheses, and then follow that with an arrow and type return type of the function.

Here is an example function that expects to be passed a callback function. Notice how we defined a type alias for the type of the callback function:

```
// @flow
// The type of the callback function used in fetchText() below
export type FetchTextCallback = (?Error, ?number, ?string) => void;

export default function fetchText(url: string, callback: FetchTextCallback) {
    let status = null;
    fetch(url)
        .then(response => {
            status = response.status;
            return response.text()
        })
        .catch(error => {
            console.error(`Error fetching ${url}: ${error}`);
        })
}
```

```

        })
        .then(body => {
            callback(null, status, body);
        })
        .catch(error => {
            callback(error, status, null);
        });
    }
}

```

17.8.10 Union Types

Let's return one more time to the `size()` function. It doesn't really make sense to have a function that does nothing other than return the length of an array. Arrays have a perfectly good `length` property for that. But `size()` might be useful if it could take any kind of collection object (an array or a Set or a Map) and return the number of elements in the collection. In regular untyped JavaScript it would be easy to write a `size()` function like that. With Flow, we need a way to express a type that allows arrays, Sets, and Maps, but doesn't allow values of any other type.

Flow calls types like this *Union types* and allows you to express them by simply listing the desired types and separating them with vertical bar characters:

```
// @flow
function size(collection: Array<mixed>|Set<mixed>|Map<mixed,mixed>): number {
    if (Array.isArray(collection)) {
        return collection.length;
    } else {
        return collection.size;
    }
}
size([1,true,"three"]) + size(new Set([true,false])) // => 5
```

Union types can be read using the word “or”—“an array or a Set or a Map”—so the fact that this Flow syntax uses the same vertical bar character as JavaScript’s OR operators is intentional.

We saw earlier that putting a question mark before a type allows `null` and `undefined` values. And now you can see that a `? prefix` is simply a shortcut for adding a `|null|void` suffix to a type.

In general, when you annotate a value with a Union type, Flow will not allow you to use that value until you’ve done enough tests to figure out what the type of the actual value is. In the `size()` example we just looked at, we need to explicitly check whether the argument is an array before we try to access the `length` property of the argument. Note that we do not have to distinguish a Set argument from a Map argument, however: both of those classes define a `size` property, so the code in the `else` clause is safe as long as the argument is not an array.

17.8.11 Enumerated Types and Discriminated Unions

Flow allows you to use primitive literals as types that consist of that one single value. If you write `let x:3;`, then Flow will not allow you to assign any value to that variable other than 3. It is not often useful to define types that have only a single member, but a union of literal types can be useful. You can probably imagine a use for types like these, for example:

```
type Answer = "yes" | "no";
type Digit = 0|1|2|3|4|5|6|7|8|9;
```

If you use types made up of literals, you need to understand that only literal values are allowed:

```
let a: Answer = "Yes".toLowerCase(); // Error: can't assign string to Answer
let d: Digit = 3+4; // Error: can't assign number to Digit
```

When Flow checks your types, it does not actually do the calculations: it just checks the types of the calculations. Flow knows that `toLowerCase()` returns a string and that the `+` operator on numbers returns a number. Even though we know that both of these calculations return values that are within the type, Flow cannot know that and flags errors on both of these lines.

A union type of literal types like `Answer` and `Digit` is an example of an *enumerated type*, or *enum*. A canonical use case for enum types is to represent the suits of playing cards:

```
type Suit = "Clubs" | "Diamonds" | "Hearts" | "Spades";
```

A more relevant example might be HTTP status codes:

```
type HttpStatus =
  | 200 // OK
  | 304 // Not Modified
  | 403 // Forbidden
  | 404; // Not Found
```

One of the pieces of advice that new programmers often hear is to avoid using literals in their code and to instead define symbolic constants to represent those values. One practical reason for this is to avoid the problem of typos: if you misspell a string literal like “Diamonds” JavaScript may never complain but your code may not work right. If you mistype an identifier, on the other hand, JavaScript is likely to throw an error that you’ll notice. With Flow, this advice does not always apply. If you annotate a variable with the type `Suit`, and then try to assign a misspelled suit to it, Flow will alert you to the error.

Another important use for literal types is the creation of *discriminated unions*. When you work with union types (made up of actually different types, not of literals), you typically have to write code to discriminate among the possible types. In the previous section, we wrote a function that could take an array or a Set or a Map as its argument and had to write code to discriminate array input from Set or Map input. If you want to create a union of Object types, you can make these types easy to discriminate by using a literal type within each of the individual Object types.

An example will make this clear. Suppose you’re using a worker thread in Node ([§16.11](#)) and are using `postMessage()` and “message” events for sending object-based messages between the main thread and the worker thread. There are multiple types of messages that the worker might want to send to the main thread, but we’d like to write a Flow Union type that describes all possible messages. Consider this code:

```
// @flow
// The worker sends a message of this type when it is done
// reticulating the splines we sent it.
export type ResultMessage = {
  messageType: "result",
  result: Array<ReticulatedSpline>, // Assume this type is defined elsewhere.
};

// The worker sends a message of this type if its code failed with an exception.
export type ErrorMessage = {
```

```

        messageType: "error",
        error: Error,
    };

    // The worker sends a message of this type to report usage statistics.
    export type StatisticsMessage = {
        messageType: "stats",
        splinesReticulated: number,
        splinesPerSecond: number
    };

    // When we receive a message from the worker it will be a WorkerMessage.
    export type WorkerMessage = ResultMessage | ErrorMessage | StatisticsMessage;

    // The main thread will have an event handler function that is passed
    // a WorkerMessage. But because we've carefully defined each of the
    // message types to have a messageType property with a literal type,
    // the event handler can easily discriminate among the possible messages:
    function handleMessageFromReticulator(message: WorkerMessage) {
        if (message.messageType === "result") {
            // Only ResultMessage has a messageType property with this value
            // so Flow knows that it is safe to use message.result here.
            // And Flow will complain if you try to use any other property.
            console.log(message.result);
        } else if (message.messageType === "error") {
            // Only ErrorMessage has a messageType property with value "error"
            // so knows that it is safe to use message.error here.
            throw message.error;
        } else if (message.messageType === "stats") {
            // Only StatisticsMessage has a messageType property with value "stats"
            // so knows that it is safe to use message.splinesPerSecond here.
            console.info(message.splinesPerSecond);
        }
    }
}

```

17.9 Summary

JavaScript is the most-used programming language in the world today. It is a living language—one that continues to evolve and improve—surrounded by a flourishing ecosystem of libraries, tools, and extensions. This chapter introduced some of those tools and extensions, but there are many more to learn about. The JavaScript ecosystem flourishes because the JavaScript developer community is active and vibrant, full of peers who share their knowledge through blog posts, videos, and conference presentations. As you close this book and go forth to join this community, you will find no shortage of information sources to keep you engaged with and learning about JavaScript.

Best wishes, David Flanagan, March 2020

¹ If you have programmed with Java, you may have experienced something like this the first time you wrote a generic API that used a type parameter. I found the learning process for Flow to be remarkably similar to what I went through in 2004 when generics were added to Java.