ES6 Essentials

This is a non-exhaustive list of new syntactic features and methods that were added to JavaScript in ES6. These features are the most commonly used and most helpful.

Prefer `const` and `let` over `var`

If you've worked with ES5 JavaScript before, you're likely used to seeing variables declared with `var`:

es6/const_let.js

```
var myVariable = 5;
```

Both the `const` and `let` statements also declare variables. They were introduced in ES6.

Use `const` in cases where a variable is never re-assigned. Using `const` makes this clear to whoever is reading your code. It refers to the "constant" state of the variable in the context it is defined within.

If the variable will be re-assigned, use `let`.

We encourage the use of `const` and `let` instead of `var`. In addition to the restriction introduced by `const`, both `const` and `let` are *block scoped* as opposed to *function scoped*. This scoping can help avoid unexpected bugs.

Arrow functions

There are three ways to write arrow function bodies. For the examples below, let's say we have an array of city objects:

es6/arrow_funcs.js

```
const cities = [
  { name: 'Cairo', pop: 7764700 },
  { name: 'Lagos', pop: 8029200 },
];
```

If we write an arrow function that spans multiple lines, we must use braces to delimit the function body like this:

appendix/es6/arrow_funcs.js

```
const formattedPopulations = cities.map((city) => {
  const popMM = (city.pop / 1000000).toFixed(2);
  return popMM + ' million';
});
console.log(formattedPopulations);
// -> [ "7.76 million", "8.03 million" ]
```

Note that we must also explicitly specify a `return` for the function.

However, if we write a function body that is only a single line (or single expression) we can use parentheses to delimit it:

appendix/es6/arrow_funcs.js

```
const formattedPopulations2 = cities.map((city) => (
  (city.pop / 1000000).toFixed(2) + ' million'
));
```

Notably, we don't use `return` as it's implied.

Furthermore, if your function body is terse you can write it like so:

appendix/es6/arrow_funcs.js

```
const pops = cities.map(city => city.pop);
console.log(pops);
// [ 7764700, 8029200 ]
```

The terseness of arrow functions is one of two reasons that we use them. Compare the one-liner above to this:

```
const popsNoArrow = cities.map(function(city) { return city.pop });
```

Of greater benefit, though, is how arrow functions bind the `this` object.

The traditional JavaScript function declaration syntax (`function () {}`) will bind `this` in anonymous functions to the global object. To illustrate the confusion this causes, consider the following example:

appendix/es6/arrow_funcs_jukebox_1.js

```
function printSong() {
  console.log("Oops - The Global Object");
}

const jukebox = {
  songs: [
    {
      title: "Wanna Be Startin' Somethin'",
      artist: "Michael Jackson",
    },
    {
      title: "Superstar",
      artist: "Madonna",
    },
  ],
  printSong: function (song) {
    console.log(song.title + " - " + song.artist);
  },
  printSongs: function () {
    // `this` bound to the object (OK)
    this.songs.forEach(function(song) {
      // `this` bound to global object (bad)
      this.printSong(song);
    });
  },
}

jukebox.printSongs();
// > "Oops - The Global Object"
// > "Oops - The Global Object"
```

The method `printSongs()` iterates over `this.songs` with `forEach()`. In this context, `this` is bound to the object (`jukebox`) as expected. However, the anonymous function passed to `forEach()` binds its internal `this` to the global object. As such, `this.printSong(song)` calls the function declared at the top of the example, *not* the method on `jukebox`.

JavaScript developers have traditionally used workarounds for this behavior, but arrow functions solve the problem by **capturing the `this` value of the enclosing context**. Using an arrow function for `printSongs()` has the expected result:

appendix/es6/arrow_funcs_jukebox_2.js

```
  printSongs: function () {
    this.songs.forEach((song) => {
      // `this` bound to same `this` as `printSongs()` (`jukebox`)
      this.printSong(song);
    });
  },
}

jukebox.printSongs();
// > "Wanna Be Startin' Somethin' - Michael Jackson"
// > "Superstar - Madonna"
```

For this reason, throughout the book we use arrow functions for all anonymous functions.

Modules

ES6 formally supports modules using the `import`/`export` syntax.

Named exports

Inside any file, you can use `export` to specify a variable the module should expose. Here's an example of a file that exports two functions:

```
// greetings.js

export const sayHi = () => (console.log('Hi!'));
export const sayBye = () => (console.log('Bye!'));

const saySomething = () => (console.log('Something!'));
```

Now, anywhere we wanted to use these functions we could use `import`. We need to specify which functions we want to import. A common way of doing this is using ES6's destructuring assignment syntax to list them out like this:

```
// app.js

import { sayHi, sayBye } from './greetings';

sayHi(); // -> Hi!
sayBye(); // => Bye!
```

Importantly, the function that was *not* exported (`saySomething`) is unavailable outside of the module.

Also note that we supply a **relative path** to `from`, indicating that the ES6 module is a local file as opposed to an npm package.

Instead of inserting an `export` before each variable you'd like to export, you can use this syntax to list off all the exposed variables in one area:

```
// greetings.js

const sayHi = () => (console.log('Hi!'));
const sayBye = () => (console.log('Bye!'));

const saySomething = () => (console.log('Something!'));

export { sayHi, sayBye };
```

We can also specify that we'd like to import all of a module's functionality underneath a given namespace with the `import * as <Namespace>` syntax:

```
// app.js

import * as Greetings from './greetings';

Greetings.sayHi();
  // -> Hi!
Greetings.sayBye();
  // => Bye!
Greetings.saySomething();
  // => TypeError: Greetings.saySomething is not a function
```

Default export

The other type of export is a default export. A module can only contain one default export:

```
// greetings.js

const sayHi = () => (console.log('Hi!'));
const sayBye = () => (console.log('Bye!'));

const saySomething = () => (console.log('Something!'));

const Greetings = { sayHi, sayBye };

export default Greetings;
```

This is a common pattern for libraries. It means you can easily import the library wholesale without specifying what individual functions you want:

```
// app.js

import Greetings from './greetings';

Greetings.sayHi(); // -> Hi!
Greetings.sayBye(); // => Bye!
```

It's not uncommon for a module to use a mix of both named exports and default exports. For instance, with `react-dom`, you can import `ReactDOM` (a default export) like this:

```
import ReactDOM from 'react-dom';

ReactDOM.render(
  // ...
);
```

Or, if you're only going to use the `render()` function, you can import the named `render()` function like this:

```
import { render } from 'react-dom';

render(
  // ...
);
```

To achieve this flexibility, the export implementation for `react-dom` looks something like this:

```
// a fake react-dom.js

export const render = (component, target) => {
  // ...
};

const ReactDOM = {
  render,
  // ... other functions
};

export default ReactDOM;
```

If you want to play around with the module syntax, check out the folder `code/webpack/es6-modules`.

For more reading on ES6 modules, see this article from Mozilla: "ES6 in Depth: Modules[196]".

`Object.assign()`

We use `Object.assign()` often throughout the book. We use it in areas where we want to create a modified version of an existing object.

`Object.assign()` accepts any number of objects as arguments. When the function receives two arguments, it *copies* the properties of the second object onto the first, like so:

appendix/es6/object_assign.js

```
const coffee = { };
const noCream = { cream: false };
const noMilk = { milk: false };
Object.assign(coffee, noCream);
// coffee is now: `{ cream: false }`
```

It is idiomatic to pass in three arguments to `Object.assign()`. The first argument is a new JavaScript object, the one that `Object.assign()` will ultimately return. The second is the object whose properties we'd like to build off of. The last is the changes we'd like to apply:

[196] https://hacks.mozilla.org/2015/08/es6-in-depth-modules/

es6/object_assign.js

```
const coffeeWithMilk = Object.assign({}, coffee, { milk: true });
// coffeeWithMilk is: `{ cream: false, milk: true }`
// coffee was not modified: `{ cream: false, milk: false }`
```

`Object.assign()` is a handy method for working with "immutable" JavaScript objects.

Template literals

In ES5 JavaScript, you'd interpolate variables into strings like this:

appendix/es6/template_literals_1.js

```
var greeting = 'Hello, ' + user + '! It is ' + degF + ' degrees outside.';
```

With ES6 template literals, we can create the same string like this:

appendix/es6/template_literals_2.js

```
const greeting = `Hello, ${user}! It is ${degF} degrees outside.`;
```

The spread operator (. . .)

In arrays, the ellipsis . . . operator will *expand* the array that follows into the parent array. The spread operator enables us to succinctly construct new arrays as a composite of existing arrays.

Here is an example:

appendix/es6/spread_operator_arrays.js

```
const a = [ 1, 2, 3 ];
const b = [ 4, 5, 6 ];
const c = [ ...a, ...b, 7, 8, 9 ];

console.log(c);  // -> [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

Notice how this is different than if we wrote:

es6/spread_operator_arrays.js

```
const d = [ a, b, 7, 8, 9 ];
console.log(d); // -> [ [ 1, 2, 3 ], [ 4, 5, 6 ], 7, 8, 9 ]
```

Enhanced object literals

In ES5, all objects were required to have explicit key and value declarations:

appendix/es6/enhanced_object_literals.js

```
const explicit = {
  getState: getState,
  dispatch: dispatch,
};
```

In ES6, you can use this terser syntax whenever the property name and variable name are the same:

appendix/es6/enhanced_object_literals.js

```
const implicit = {
  getState,
  dispatch,
};
```

Lots of open source libraries use this syntax, so it's good to be familiar with it. But whether you use it in your own code is a matter of stylistic preference.

Default arguments

With ES6, you can specify a default value for an argument in the case that it is `undefined` when the function is called.

This:

es6/default_args.js

```
function divide(a, b) {
  // Default divisor to `1`
  const divisor = typeof b === 'undefined' ? 1 : b;

  return a / divisor;
}
```

Can be written as this:

appendix/es6/default_args.js

```
function divide(a, b = 1) {
  return a / b;
}
```

In both cases, using the function looks like this:

appendix/es6/default_args.js

```
divide(14, 2);
// => 7
divide(14, undefined);
// => 14
divide(14);
// => 14
```

Whenever the argument `b` in the example above is `undefined`, the default argument is used. Note that `null` will not use the default argument:

appendix/es6/default_args.js

```
divide(14, null); // `null` is used as divisor
// => Infinity    // 14 / null
```

Destructuring assignments

For arrays

In ES5, extracting and assigning multiple elements from an array looked like this:

es6/destructuring_assignments.js

```
var fruits = [ 'apples', 'bananas', 'oranges' ];
var fruit1 = fruits[0];
var fruit2 = fruits[1];
```

In ES6, we can use the destructuring syntax to accomplish the same task like this:

appendix/es6/destructuring_assignments.js

```
const [ veg1, veg2 ] = [ 'asparagus', 'broccoli', 'onion' ];
console.log(veg1); // -> 'asparagus'
console.log(veg2); // -> 'broccoli'
```

The variables in the array on the left are "matched" and assigned to the corresponding elements in the array on the right. Note that `'onion'` is ignored and has no variable bound to it.

For objects

We can do something similar for extracting object properties into variables:

appendix/es6/destructuring_assignments.js

```
const smoothie = {
  fats: [ 'avocado', 'peanut butter', 'greek yogurt' ],
  liquids: [ 'almond milk' ],
  greens: [ 'spinach' ],
  fruits: [ 'blueberry', 'banana' ],
};

const { liquids, fruits } = smoothie;

console.log(liquids); // -> [ 'almond milk' ]
console.log(fruits); // -> [ 'blueberry', 'banana' ]
```

Parameter context matching

We can use these same principles to bind arguments inside a function to properties of an object supplied as an argument:

es6/destructuring_assignments.js

```
const containsSpinach = ({ greens }) => {
  if (greens.find(g => g === 'spinach')) {
    return true;
  } else {
    return false;
  }
};

containsSpinach(smoothie); // -> true
```

We do this often with functional React components:

appendix/es6/destructuring_assignments.js

```
const IngredientList = ({ ingredients, onClick }) => (
  <ul className='IngredientList'>
    {
      ingredients.map(i => (
        <li
          key={i.id}
          onClick={() => onClick(i.id)}
          className='item'
        >
          {i.name}
        </li>
      ))
    }
  </ul>
)
```

Here, we use destructuring to extract the props into variables (`ingredients` and `onClick`) that we then use inside the component's function body.