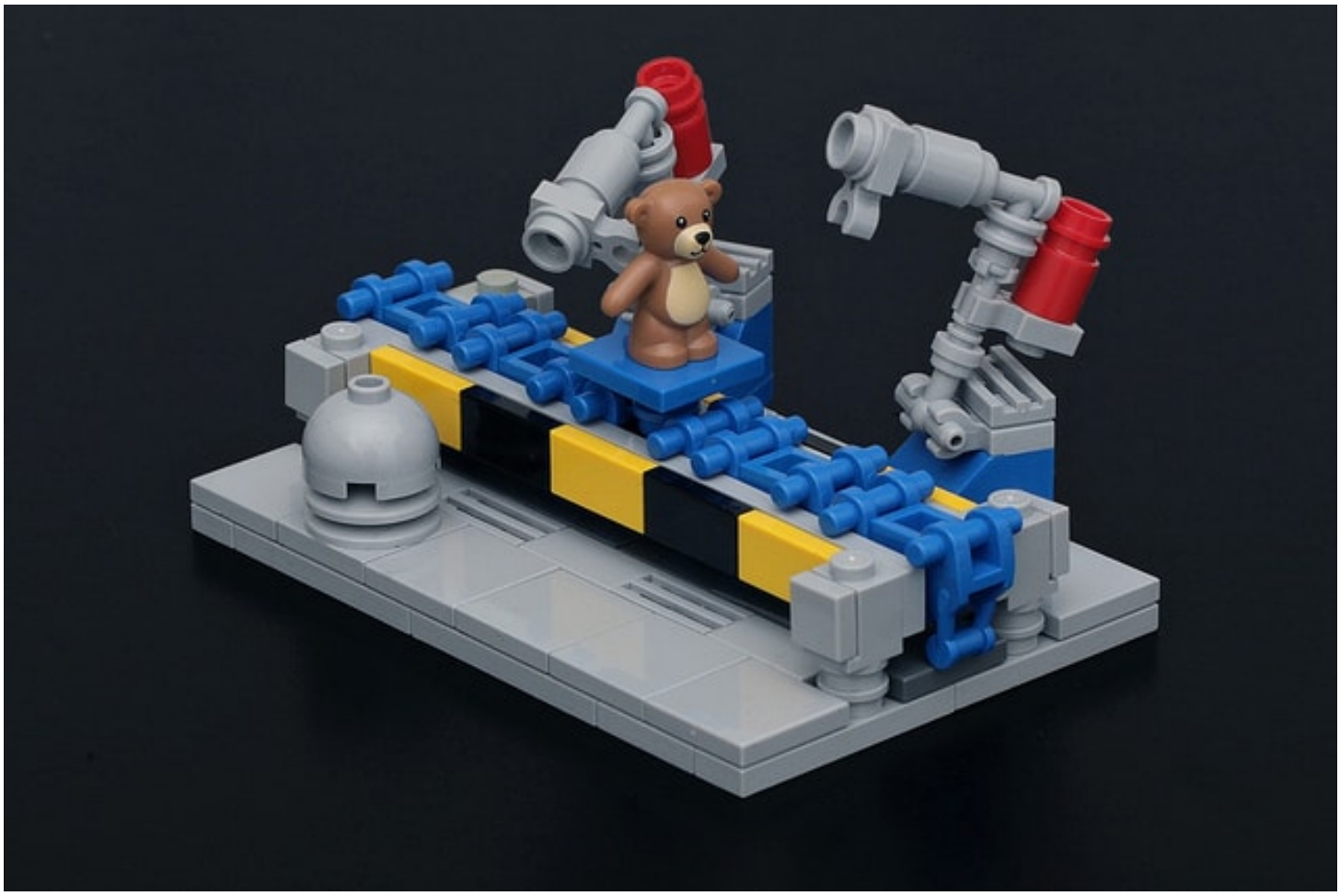

Using ES6 arrow functions in production-ready apps

production ready apps

Published under **ECMASCRIPT**

If you are just getting started with ES6, you might have heard about "fat arrow functions". They are a great addition to the ECMAScript 6 standard and their origin is probably the CoffeeScript function declaration. You can think of them as throwaway functions that a you can attach to a click or mouse event. There are a couple of ways to use arrow functions and we are going to go over each one in turn.

Table of contents

Anonymous callbacks

The first and most intuitive use for an arrow function would be as an anonymous callback. You can attach an arrow function to a button's click event as but you can also pass it as a callback to **`Array.prototype.map`**.

```
// click.js
const btn = document.querySelector('button');

btn.addEventListener('click', (event) => {
    console.log('clicked');
});
```

Every time we click the button identified by the **`btn`** variable, we are going to log the message "clicked". Nothing too fancy, just an anonymous function

responding to an event.

```
// map.js
let numbers = [1,2,3,4];
let multipliedBy2 = numbers.map((number) => {
    return number * 2;
});

console.log(multipliedBy2); // [2, 4, 6, 8]
```

As I mentioned in my ES6 arrow functions in depth article, the arrow function implicitly returns the result of executing its logic, if it has a concise body / block body. This feature makes the arrow function an ideal callback for **`Array`** operations that return new arrays, like **`map`** and **`filter`**.

With this in mind, let's simplify our **`map`** example.

```
const numbers = [1, 2, 3, 4];
const multipliedBy2 = numbers.map((n) => n * 2);

// or without the parens around the argument
const multipliedBy2WithShorterCallback = numbers.map(n => n * 2);
```

Function expressions

The function expression form of the arrow function is very popular in the React.js community. This is mostly due to its concise semantics. One of the people making extensive use of this feature in their examples and tutorials is Dan Abramov, the author of Redux

Dan Abramov, the author of Redux.

```
const todos = (state = [], action) => {
    switch(action.type) {

    case ADD_TODO:
        return [..state, action.text];
    default:
        return state;
    }
}
```

This allows you to call your otherwise anonymous arrow function wherever you need it, by using the **todos** identifier. Just be careful that the hoisting for function expressions is different from the regular function. While the function is hoisted with all its body, to the top of the scope, function expressions are treated as regular variables, and only the **todos** identifier will be hoisted to the top of the scope but without the function body. This is why you cannot call functions defined using function expressions, before the site where they are assigned to the identifier in the code.

Returning data

If you take a look at the **map** example, you can tell from afar that it is very easy to return data from an arrow function.

Using the `return` keyword

As with any other function, you can return data from an arrow function using the **return** keyword.

```
const multiply = (x) => {
    return (y) => {
        return x * y;
    }
```

```
};

const multiplyBy3 = multiply(3);


multiplyBy3(2); // 6
multiplyBy3(3); // 9
```

Using the implicit `return`

Let's rewrite our example to use the implicit return feature of arrow functions.

```
const multiply = (x) => (y) => x * y;

const multiplyBy3 = multiply(3);
multiplyBy3(2); // 6
multiplyBy3(3); // 9
```

If you need to return objects from an arrow functions, there's a catch. You cannot use the curly braces directly as that would throw an error.

```
const getInitialData = () => {
    id: 1,
    name: 'Jane Doe'
};

// This will throw an error
let initialData = getInitialData();
```

The JavaScript engine expects the object's curly braces to be a block of code. Instead it finds identifiers, colons and commas and it doesn't quite know what to do with them.

To work around this, all you need to do is to surround your whole object in parenthesis. This will not throw an error and your code will work as expected and also look good in the process.

```
// This is perfectly valid
const getInitialData = () => ({
    id: 1,
    name: 'Jane Doe'
});

let initialData = getInitialData();
```

With this last part, we wrapped up all known usages of arrow functions in real-life, production apps. For more, in-depth knowledge on the subject, refer to my ES6 arrow functions in depth article, where I talk more about the internals of the arrow function.

> *Photo credits: Pascal — Killbot Assembly Line*

JOIN →

Previous Article:

ES6 arrow functions in depth

Next Article:

Top 5 things to consider when choosing a new technology

Let's work together.

Have a project in mind? I'd love to hear more about it.

CONTACT ME →