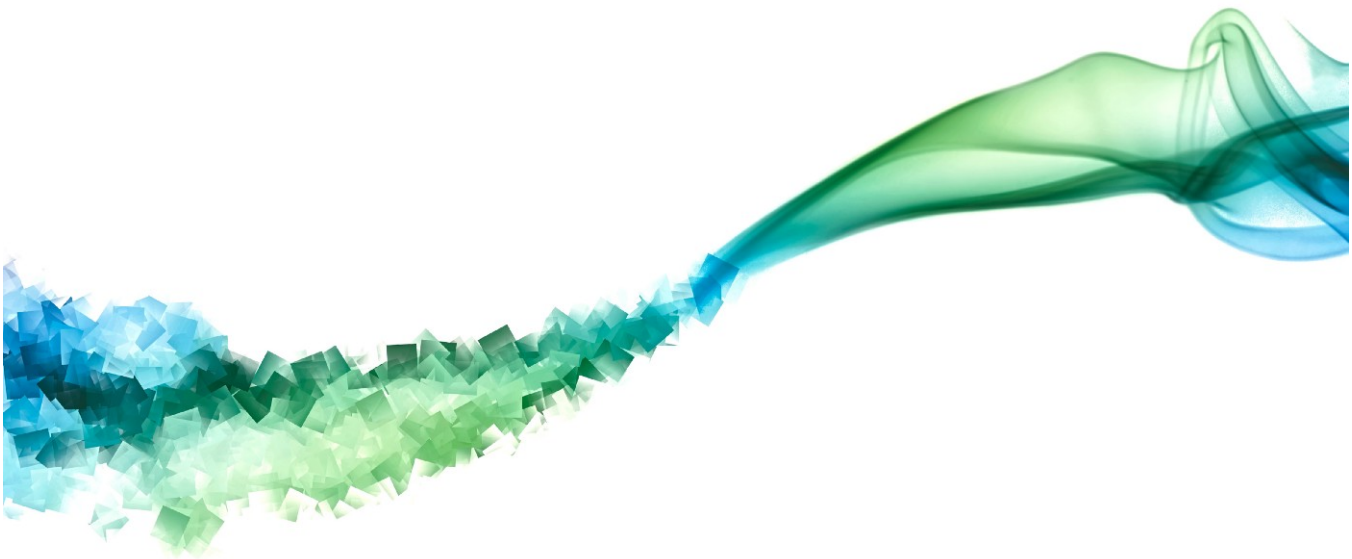
JavaScript Factory Functions with ES6+

Eric Elliott
Jul 20, 2017 · 10 min read



Smoke Art Cubes to Smoke — MattysFlicks — (CC BY 2.0)

> ***Note:*** *This is part of the "Composing Software" series* ***(now a book!)*** *on learning functional programming and compositional software techniques in JavaScript ES6+ from the ground up. Stay tuned. There's a lot more of this to come!*
> *Buy the Book | Index | < Previous | Next >*

A **factory function** is any function which is not a class or constructor that returns a (presumably new) object. In JavaScript, any function can return an object. When it does so without the `new` keyword, it's a factory function.

Factory functions have always been attractive in JavaScript because they offer the ability to easily produce object instances without diving into the complexities of classes and the `new` keyword.

JavaScript provides a very handy object literal syntax. It looks something like this:

```
const user = {
  userName: 'echo',
  avatar: 'echo.png'
};
```

Like JSON (which is based on JavaScript's object literal notation), the left side of the `:` is the property name, and the right side is the value. You can access props with dot notation:

```
console.log(user.userName); // "echo"
```

You can access computed property names using square bracket notation:

```
const key = 'avatar';

console.log( user[key] ); // "echo.png"
```

If you have variables in-scope with the same name as your intended property names, you can omit the colon and the value in the object literal creation:

```
const userName = 'echo';
const avatar = 'echo.png';

const user = {
  userName,
  avatar
};

console.log(user);
// { "avatar": "echo.png",   "userName": "echo" }
```

Object literals support concise method syntax. We can add a `.setUserName()` method:

```
const userName = 'echo';
const avatar = 'echo.png';
```

```
const user = {
  userName,
  avatar,

  setUserName (userName) {
    this.userName = userName;
    return this;
  }
};

console.log(user.setUserName('Foo').userName); // "Foo"
```

In concise methods, `this` refers to the object which the method is called on. To call a method on an object, simply access the method using object dot notation and invoke it by using parentheses, e.g., `game.play()` would apply `.play()` to the `game` object. In order to apply a method using dot notation, that method must be a property of the object in question. You can also apply a method to any other arbitrary object using the function prototype methods, `.call()`, `.apply()`, or `.bind()`.

In this case, `user.setUserName('Foo')` applies `.setUserName()` to `user`, so `this === user`. In the `.setUserName()` method, we change the `.userName` property on the `user` object via its `this` binding, and return the same object instance for method chaining.

Literals for One, Factories for Many

If you need to create many objects, you'll want to combine the power of object literals and factory functions.

With a factory function, you can create as many user objects as you want. If you're building a chat app, for instance, you can have a user object representing the current user, and also a lot of other user objects representing all the other users who are currently signed in and chatting, so you can display their names and avatars, too.

Let's turn our `user` object into a `createUser()` factory:

```
const createUser = ({ userName, avatar }) => ({
  userName,
  avatar,
  setUserName (userName) {
    this.userName = userName;
    return this;
  }
});
```

```
console.log(createUser({ userName: 'echo', avatar: 'echo.png' }));

/*
{
  "avatar": "echo.png",
  "userName": "echo",
  "setUserName": [Function setUserName]
}
*/
```

Returning Objects

Arrow functions (`=>`) have an implicit return feature: if the function body consists of a single expression, you can omit the `return` keyword: `() => 'foo'` is a function that takes no parameters, and returns the string, `"foo"` .

Be careful when you return object literals. By default, JavaScript assumes you want to create a function body when you use braces, e.g., `{ broken: true }` . If you want to use an implicit return for an object literal, you'll need to disambiguate by wrapping the object literal in parentheses:

```
const noop = () => { foo: 'bar' };
console.log(noop()); // undefined

const createFoo = () => ({ foo: 'bar' });
console.log(createFoo()); // { foo: "bar" }
```

In the first example, `foo:` is interpreted as a label, and `bar` is interpreted as an expression that doesn't get assigned or returned. The function returns `undefined` .

In the `createFoo()` example, the parentheses force the braces to be interpreted as an expression to be evaluated, rather than a function body block.

Destructuring

Pay special attention to the function signature:

```
const createUser = ({ userName, avatar }) => ({
```

In this line, the braces (`{ , }`) represent object destructuring. This function takes one argument (an object), but destructures two formal parameters from that single argument, `userName` , and `avatar` . Those parameters can then be used as variables in the function body scope. You can also destructure arrays:

```
const swap = ([first, second]) => [second, first];

console.log( swap([1, 2]) ); // [2, 1]
```

And you can use the rest and spread syntax (`...varName`) to gather the rest of the values from the array (or a list of arguments), and then spread those array elements back into individual elements:

```
const rotate = ([first, ...rest]) => [...rest, first];

console.log( rotate([1, 2, 3]) ); // [2, 3, 1]
```

Computed Property Keys

Earlier we used square bracket computed property access notation to dynamically determine which object property to access:

```
const key = 'avatar';

console.log( user[key] ); // "echo.png"
```

We can also compute the value of keys to assign to:

```
const arrToObj = ([key, value]) => ({ [key]: value });

console.log( arrToObj([ 'foo', 'bar' ]) ); // { "foo": "bar" }
```

In this example, `arrToObj` takes an array consisting of a key/value pair (aka a tuple) and converts it into an object. Since we don't know the name of the key, we need to compute the property name in order to set the key/value pair on the object. For that, we

borrow the idea of square bracket notation from computed property accessors, and reuse it in the context of building an object literal:

```
{ [key]: value }
```

After the interpolation is done, we end up with the final object:

```
{ "foo": "bar" }
```

Default Parameters

Functions in JavaScript support default parameter values, which have several benefits:

1. Users are able to omit parameters with suitable defaults.
2. The function is more self-documenting because default values supply examples of expected input.
3. IDEs and static analysis tools can use default values to infer the type expected for the parameter. For example, a default value of `1` implies that the parameter can take a member of the `Number` type.

Using default parameters, we can document the expected interface for our `createUser` factory, and automatically fill in `'Anonymous'` details if the user's info is not supplied:

```
const createUser = ({
  userName = 'Anonymous',
  avatar = 'anon.png'
} = {}) => ({
  userName,
  avatar
});

console.log(
  // { userName: "echo", avatar: 'anon.png' }
  createUser({ userName: 'echo' }),

  // { userName: "Anonymous", avatar: 'anon.png' }
  createUser()
);
```

The last part of the function signature probably looks a little funny:

```
  } = {}) => ({
```

The last `= {}` bit just before the parameter signature closes means that if nothing gets passed in for this parameter, we're going to use an empty object as the default. When you try to destructure values from the empty object, the default values for properties will get used automatically, because that's what default values do: replace `undefined` with some predefined value.

Without the `= {}` default value, `createUser()` with no arguments would throw an error because you can't try to access properties from `undefined`.

Type Inference

JavaScript does not have any native type annotations as of this writing, but several competing formats have sprung up over the years to fill the gaps, including JSDoc (in decline due to the emergence of better options), Facebook's Flow, and Microsoft's TypeScript. I use rtype for documentation — a notation I find much more readable than TypeScript for functional programming.

At the time of this writing, there is no clear winner for type annotations. None of the alternatives have been blessed by the JavaScript specification, and there seem to be clear shortcomings in all of them.

Type inference is the process of inferring types based on the context in which they are used. In JavaScript, it is a very good alternative to type annotations.

If you provide enough clues for inference in your standard JavaScript function signatures, you'll get most of the benefits of type annotations with none of the costs or risks.

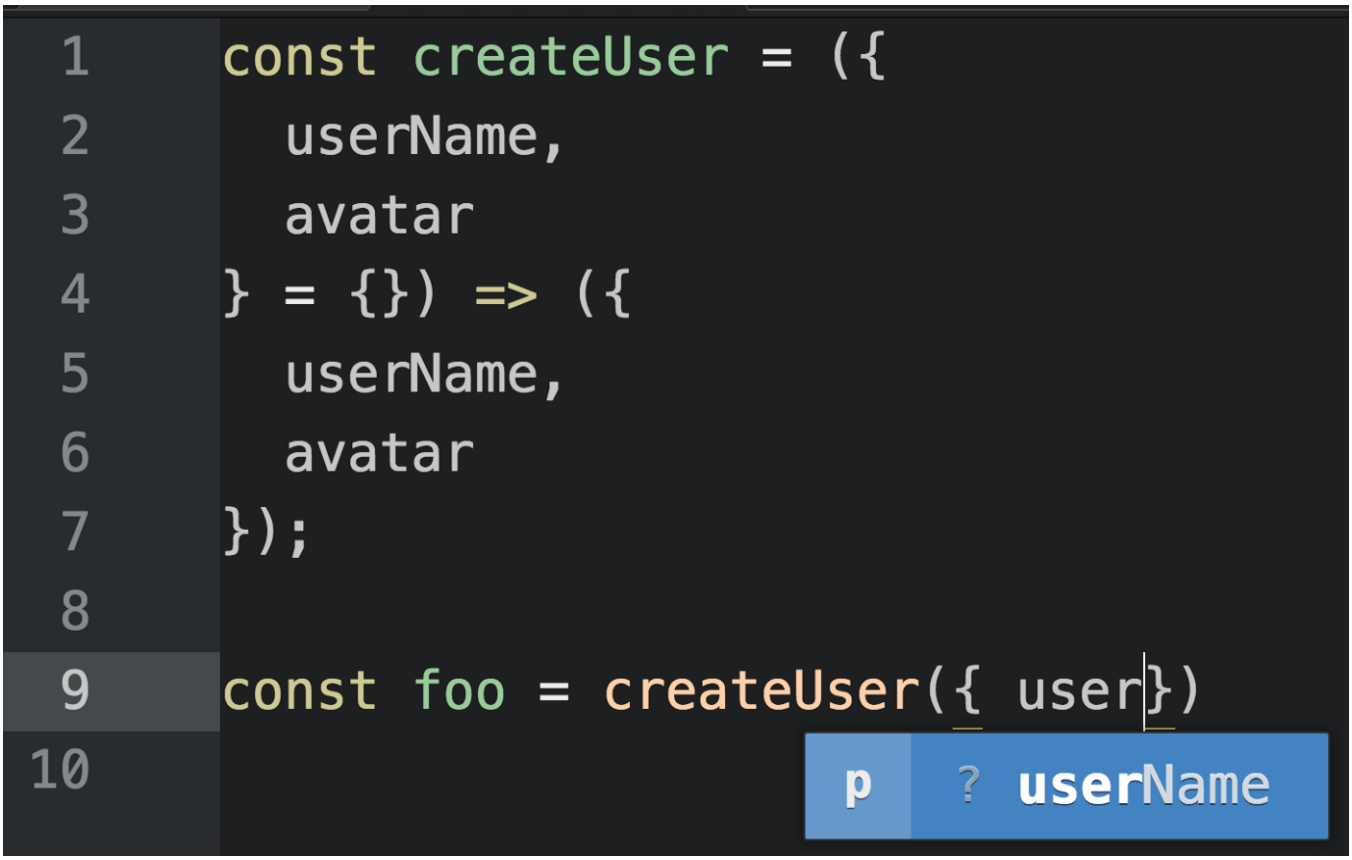
Even if you decide to use a tool like TypeScript or Flow, you should do as much as you can with type inference, and save the type annotations for situations where type inference falls short. For example, there's no native way in JavaScript to specify a shared interface. That's both easy and useful with TypeScript or rtype.

Tern.js is a popular type inference tool for JavaScript that has plugins for many code editors and IDEs.

Microsoft's Visual Studio Code doesn't need Tern because it brings the type inference capabilities of TypeScript to regular JavaScript code.

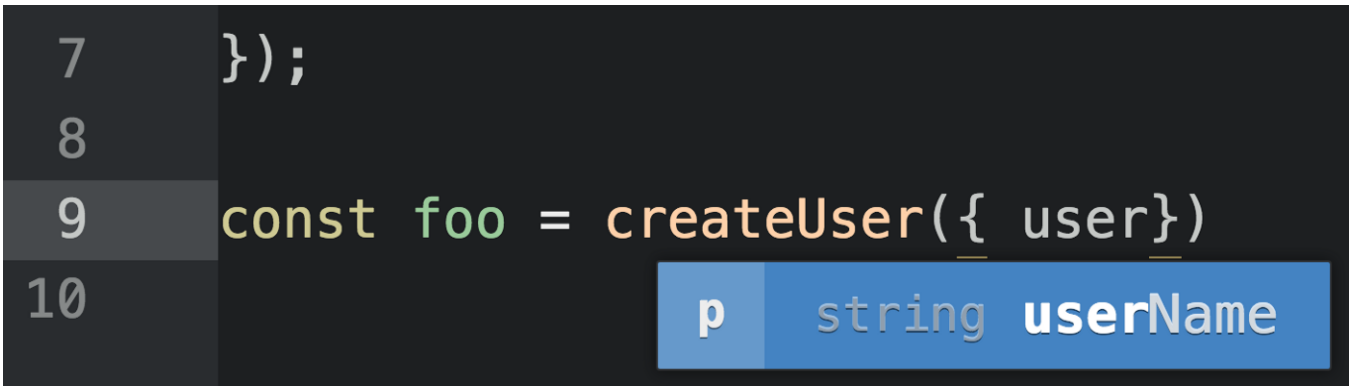
When you specify default parameters for functions in JavaScript, tools capable of type inference such as Tern.js, TypeScript, and Flow can provide IDE hints to help you use the API you're working with correctly.

Without defaults, IDEs (and frequently, humans) don't have enough hints to figure out the expected parameter type.



Without defaults, the type is unknown for `userName`.

With defaults, IDEs (and frequently, humans) can infer the types from the examples.



With defaults, the IDE can suggest that `userName` is expecting a string.

It doesn't always make sense to restrict a parameter to a fixed type (that would make generic functions and higher order functions difficult), but when it does make sense, default parameters are often the best way to do it, even if you're using TypeScript or Flow.

Factory Functions for Mixin Composition

Factories are great at cranking out objects using a nice calling API. Usually, they're all you need, but once in a while, you'll find yourself building similar features into different types of objects, and you'll want to abstract those features into functional mixins so you can reuse them more easily.

That's where functional mixins shine. Let's build a `withConstructor` mixin to add the `.constructor` property to all object instances.

`with-constructor.js`:

```
const withConstructor = constructor => o => ({
  // create the delegate [[Prototype]]
  __proto__: {
    // add the constructor prop to the new [[Prototype]]
    constructor
  },
  // mix all o's props into the new object
  ...o
});
```

Now you can import it and use it with other mixins:

```
import withConstructor from `./with-constructor';
```

```
const pipe = (...fns) => x => fns.reduce((y, f) => f(y), x);
// or `import pipe from 'lodash/fp/flow';`

// Set up some functional mixins
const withFlying = o => {
  let isFlying = false;
  return {
    ...o,
    fly () {
      isFlying = true;
      return this;
    },
    land () {
      isFlying = false;
      return this;
    },
    isFlying: () => isFlying
  }
};

const withBattery = ({ capacity }) => o => {
  let percentCharged = 100;
  return {
    ...o,
    draw (percent) {
      const remaining = percentCharged - percent;
      percentCharged = remaining > 0 ? remaining : 0;
      return this;
    },
    getCharge: () => percentCharged,
    getCapacity: () => capacity
  };
};

const createDrone = ({ capacity = '3000mAh' }) => pipe(
  withFlying,
  withBattery({ capacity }),
  withConstructor(createDrone)
)({});

const myDrone = createDrone({ capacity: '5500mAh' });

console.log(`
  can fly:  ${ myDrone.fly().isFlying() === true }
  can land: ${ myDrone.land().isFlying() === false }
  battery capacity: ${ myDrone.getCapacity() }
  battery status: ${ myDrone.draw(50).getCharge() }%
  battery drained: ${ myDrone.draw(75).getCharge() }% remaining
`);

console.log(`
  constructor linked: ${ myDrone.constructor === createDrone }
`);
```

As you can see, the reusable `withConstructor()` mixin is simply dropped into the pipeline with other mixins. `withBattery()` could be used with other kinds of objects, like robots, electric skateboards, or portable device chargers. `withFlying()` could be used to model flying cars, rockets, or air balloons.

Composition is more of a way of thinking than a particular technique in code. You can accomplish it in many ways. Function composition is just the easiest way to build it up from scratch, and factory functions are a simple way to wrap a friendly API around the implementation details.

Conclusion

ES6 provides a convenient syntax for dealing with object creation and factory functions. Most of the time, that's all you'll need, but because this is JavaScript, there's another approach that makes it feel more like Java: the `class` keyword.

In JavaScript, classes are more verbose & restrictive than factories, and a bit of a minefield when it comes to refactoring, but they've also been embraced by major front-end frameworks like React and Angular, and there are a couple of rare use-cases that make the complexity worthwhile.

> *"Sometimes, the elegant implementation is just a function. Not a method. Not a class. Not a framework. Just a function." ~ John Carmack*

Start with the simplest implementation, and move to more complex implementations only as required.

Next: Why Composition is Harder with Classes >

Next Steps

Want to learn more about object composition with JavaScript?

Learn JavaScript with Eric Elliott. If you're not a member, you're missing out!

. . .

Eric Elliott *is a distributed systems expert and author of the books, "Composing Software" and "Programming JavaScript Applications". As co-founder of DevAnywhere.io, he teaches developers the skills they need to work remotely and embrace work/life balance. He builds and advises development teams for crypto projects, and has contributed to software experiences for* ***Adobe Systems, Zumba Fitness, The Wall Street Journal, ESPN, BBC,*** *and top recording artists including* ***Usher, Frank Ocean, Metallica,*** *and many more.*

He enjoys a remote lifestyle with the most beautiful woman in the world.

Thanks to JS_Cheerleader.

JavaScript Technology