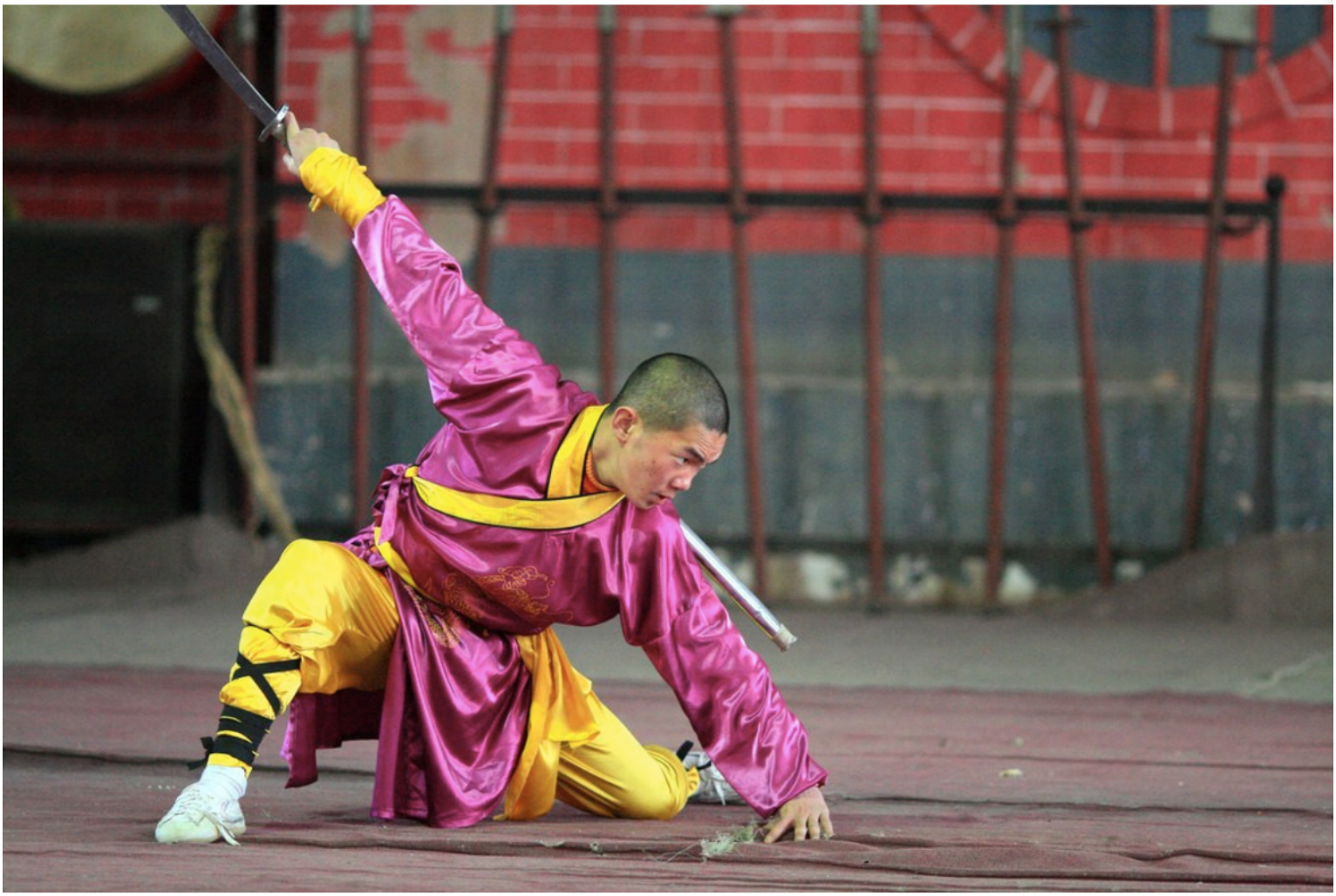
JavaScript Factory Functions vs Constructor Functions vs Classes

Eric Elliott Follow

Jun 1, 2016 · 7 min read



Shaolin — larique (CC BY-NC-ND 2.0)

Prior to ES6, there was a lot of confusion about the differences between a factory function and a constructor function in JavaScript. Since ES6 has the `*class*` keyword, a lot of people seem to think that solved many problems with constructor functions. It didn't. Let's explore the major differences you still need to be aware of.

First, let's look at examples of each:

```
// class
class ClassCar {
  drive () {
    console.log('Vroom!');
  }
}

const car1 = new ClassCar();
```

```
console.log(car1.drive());


// constructor
function ConstructorCar () {}

ConstructorCar.prototype.drive = function () {
  console.log('Vroom!');
};

const car2 = new ConstructorCar();
console.log(car2.drive());


// factory
const proto = {
  drive () {
    console.log('Vroom!');
  }
};

const factoryCar = () => Object.create(proto);

const car3 = factoryCar();
console.log(car3.drive());
```

class-constructor-factory-examples.js hosted with ❤ by GitHub view raw

Each of these strategies stores methods on a shared prototype, and optionally supports private data via constructor function closures. In other words, they have mostly the same features, and could mostly be used interchangeably.

> In JavaScript, any function can return a new object. When it's not a constructor function or class, it's called a factory function.

Top highlight

ES6 classes desugar to constructor functions, so everything that follows about constructor functions also applies to ES6 classes:

```
class Foo {}

console.log(typeof Foo); // function
```

What's the Difference Between a Factory and a Constructor?

Constructors force callers to use the `new` keyword. Factories don't. That's it, but that has some relevant side-effects.

So what does the `new` keyword do?

> *Note: We'll use `instance` to refer to the newly created instance, and `Constructor` to refer to the constructor function or class that created the instance.*

1. Instantiates a new instance object and binds `this` to it within the constructor.
2. Binds `instance.__proto__` to `Constructor.prototype`.

3. As a side-effect of 2, binds `instance.__proto__.constructor` to `Constructor`.
4. Implicitly returns `this`, which refers to `instance`.

Benefits of Constructors & `class`

- Most books teach you to use class or constructors.
- `this` refers to the new object.
- Some people like the way `myFoo = new Foo()` reads.
- There may be a micro-optimization performance benefit, but you should not worry about that unless you have profiled your code and proven that it's an issue for you.

Drawbacks of Constructors & `class`

1. Required `new`.

Prior to ES6, forgetting `new` was a very common bug. To counter it, many people used boilerplate to enforce it:

```
function Foo() {
  if (!(this instanceof Foo)) { return new Foo(); }
}
```

In ES6+ (ES2015) if you try to call a class constructor without `new`, it will always throw an error. It's not possible to avoid forcing the `new` requirement on callers without wrapping your class in a factory function.

2. Details of instantiation get leaked into the calling API (via the `new` requirement).

All callers are tightly coupled to the constructor implementation. If you ever need the additional flexibility of the factory, the refactor is a **breaking change**. Class to factory refactors are common enough that they appear in the seminal Refactoring book, "Refactoring: Improving the Design of Existing Code" by Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts.

3. Constructors break the Open / Closed Principle

Because of the `new` requirement, constructor functions violate the **open/closed** principle: an API should be **open** for extension, but **closed** for modification.

I argue that the class to factory refactor is common enough that it should be considered a standard extension for all constructors: Upgrading from a class to a factory should not break things, but in JavaScript, it does.

If you start out exporting a constructor or class and users start using the constructor, then down the road you realize you need the flexibility of a factory, (for instance, to switch the implementation to use object pools, or to instantiate across execution contexts, or to have more inheritance flexibility using alternative prototypes), you can't easily do so without

flexibility using alternative prototypes), you can't easily do so without forcing a refactor on callers.

Unfortunately, in JavaScript, switching from a constructor or class to a factory is a **breaking change**:

In the example above, we start out with a class, but we want to add the capability to offer different kinds of car bundles. To do so, the factory uses alternative prototypes for different car bundles. I've used this technique to store various implementations of a media player interface, picking the correct prototype based on the type of media the player needed to control.

4. Using Constructors Enables the Deceptive `instanceof`

One of the breaking changes in the constructor to factory refactor is `instanceof`. Sometimes people are tempted to use `instanceof` as a type check guard in their code. That can be very problematic. I recommend that you avoid `instanceof`.

> `instanceof` lies.

`instanceof` does not do type checking the way that you expect similar checks to do in strongly typed languages. Instead, it does an identity check comparing the object's `__proto__` object to the `Constructor.prototype` property.

It won't work across different memory realms like iframes, for example (a common source of bugs in 3rd party JavaScript embeds). It also doesn't work if your `Constructor.prototype` gets replaced.

It will also fail if you start out with a class or constructor (which returns `this`, linked to the `Constructor.prototype`), and then switch to exporting an arbitrary object (not linked to the `Constructor.prototype`), which is what happens when you change from a constructor to a factory.

In short, `instanceof` is another way in which switching from a constructor to a factory is a **breaking change**.

Benefits of Using class

- Convenient, self-contained syntax.
- A single, canonical way to emulate classes in JavaScript. Prior to ES6, there were several competing implementations in popular libraries.
- More familiar to people from a class-based language background.

Drawbacks of Using class

All of the constructor drawbacks, plus:

- Temptation for users to create problematic class hierarchies using the `extends` keyword.

Class hierarchies lead to a bunch of well-known problems in object oriented design, including **the fragile base class problem, the gorilla banana problem, the duplication by necessity problem,** and so on. Unfortunately, class affords extends like balls afford throwing and chairs afford sitting. For more, read "The Two Pillars of JavaScript: Prototypal OO" and "Inside the Dev Team Death Spiral".

It's worth noting that both constructors and factories can also be used to create problematic inheritance hierarchies, but with the `*extends*` keyword, class creates an affordance that leads you down the wrong path. In other words, it encourages you to think in terms of inflexible (and often wrong) *is-a* relationships, rather than the more flexible compositional *has-a* or *can-do* relationships.

> An **affordance** is a feature that affords the opportunity to perform a certain action. For example, a knob affords twisting, a lever affords pulling, a button affords pressing, etc…

Benefits of Using Factories

Factories are much more flexible than either constructor functions or classes, and they don't lead people down the wrong path by tempting them with the `*extends*` keyword and deep inheritance hierarchies. There are many safer code reuse mechanisms you should favor over class inheritance, including functions and modules.

1. Return any arbitrary object and use any arbitrary prototype

For example, you can easily create various types of objects which implement the same API, e.g., a media player that can instantiate players for multiple types of video content which use different APIs under the hood, or an event library which can emit DOM events or web socket events.

Factories can also instantiate objects across execution contexts, take advantage of object pools, and allow for more flexible prototypal inheritance models.

2. No refactoring worries

You'd never have a need to convert from a factory to a constructor, so refactoring will never be an issue.

3. No `new`

No ambiguity about using `new`. Don't. (It will make `this` behave badly, see next point).

4. Standard `this` behavior

`this` behaves as it normally would, so you can use it to access the parent object. For example, inside `player.create()`, `this` refers to player, just like any other method invocation would. `call()` and `apply()` also reassign `this` as expected.

5. No deceptive `instanceof`

6. Some people like the way `myFoo = createFoo()` reads

Drawbacks of Factories

- Doesn't create a link from the instance to `Factory.prototype` — but this is actually a good thing because you won't get a deceptive `instanceof`. Instead, `instanceof` will always fail. See benefits.
- `this` doesn't refer to the new object inside the factory. See benefits.
- It **may** perform slower than a constructor function in micro-optimization benchmarks. The slow path is still very fast — millions of ops/sec on an old computer. This is more likely to be a concern in library or framework code than in application code. Always benchmark from the user-perspective before using micro-optimizations.

Conclusion

In my opinion, `class` may have a convenient syntax, but that can't make up for the fact that it lures unwary users to crash on the rocks of class inheritance. It's also risky because in the future, you may want to upgrade to a factory, but all your callers will be tightly coupled to the constructor function because of the `new` keyword and the fact that moving from classes to factories is a breaking change.

You may be thinking that you can just refactor the call sites, but on large teams, or if the class you're working with is part of a public API, you could break code that isn't in your control. In other words, you can't always assume that refactoring callers is even an option.

The cool thing about factories is that they're not just more powerful and more flexible, they're also the easiest way to encourage entire teams, and entire API user bases to use patterns that are simple, flexible, and safe.

. . .

https://devanywhere.io/

. . .

Eric Elliott *is the author of "Programming JavaScript Applications" (O'Reilly), and cofounder of DevAnywhere.io. He 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 works anywhere he wants with the most beautiful woman in the world.

Thanks to JS_Cheerleader.

JavaScript Programming Technology

5.4K claps 35 responses

WRITTEN BY
Eric Elliott
Make some magic. #JavaScript

Follow

JavaScript Scene
JavaScript, software leadership, software development, and related technologies.

Follow