Eric Elliott Follow
Compassionate entrepreneur on a mission to end homelessness.
Jan 18, 2016 · 9 min read

Master the JavaScript Interview: What's the Difference Between Class & Prototypal Inheritance?

Electric Guitar — Feliciano Guimarães (CC BY 2.0)

> *"Master the JavaScript Interview" is a series of posts designed to prepare candidates for common questions they are likely to encounter when applying for a mid to senior-level JavaScript position. These are questions I*

> *frequently use in real interviews. Want to start from the beginning? See “What is a Closure?”*

> *Note: This article uses ES6 examples. If you haven’t learned ES6 yet, see “How to Learn ES6”.*

Objects are frequently used in JavaScript, and understanding how to work with them effectively will be a huge win for your productivity. In fact, poor OO design can potentially lead to project failure, and in the worst cases, company failures.

Unlike most other languages, JavaScript’s object system is based on **prototypes, not classes**. Unfortunately, most JavaScript developers don’t understand JavaScript’s object system, or how to put it to best use. Others do understand it, but want it to behave more like class based systems. The result is that JavaScript’s object system has a confusing split personality, which means that JavaScript developers need to know a bit about **both prototypes and classes**.

What’s the Difference Between Class & Prototypal Inheritance?

This can be a tricky question, and you’ll probably need to defend your answer with follow-up Q&A, so pay special attention to learning the differences, and how to apply the knowledge to write better code.

Class Inheritance: ***A class is like a blueprint—a description of the object to be created.*** Classes inherit from classes and **create subclass relationships**: hierarchical class taxonomies.

Instances are typically instantiated via constructor functions with the `` `new` `` keyword. Class inheritance may or may not use the `` `class` `` keyword from ES6. Classes as you may know them from languages like Java don’t technically exist in JavaScript. Constructor functions are used, instead. The ES6 `` `class` `` keyword desugars to a constructor function:

```
class Foo {}
typeof Foo // 'function'
```

In JavaScript, class inheritance is implemented on top of prototypal inheritance, but *that does not mean that it does the same thing:*

JavaScript's class inheritance uses the prototype chain to wire the child `` `Constructor.prototype` `` to the parent `` `Constructor.prototype` `` for delegation. Usually, the `` `super()` `` constructor is also called. Those steps form **single-ancestor parent/child hierarchies** and ***create the tightest coupling available in OO design.***

> *"Classes inherit from classes and create subclass relationships: hierarchical class taxonomies."*

Prototypal Inheritance: *A prototype is a working object instance.* Objects inherit directly from other objects.

Instances may be composed from many different source objects, allowing for easy selective inheritance and a flat [[Prototype]] delegation hierarchy. In other words, **class taxonomies are not an automatic side-effect of prototypal OO**: *a critical distinction.*

Instances are typically instantiated via factory functions, object literals, or `` `Object.create()` ``.

> *"A prototype is a working object instance. Objects inherit directly from other objects."*

Why Does this Matter?

Inheritance is fundamentally a code reuse mechanism: A way for different kinds of objects to share code. The way that you share code matters because if you get it wrong, **it can create a lot of problems,** specifically:

Class inheritance creates parent/child object taxonomies as a side-effect.

Those taxonomies are virtually impossible to get right for all new use cases, and widespread use of a base class leads to **the fragile base class problem,** which makes them difficult to fix when you get them wrong.

In fact, class inheritance causes many well known problems in OO design:

- **The tight coupling problem** (class inheritance is the tightest coupling available in oo design), which leads to the next one…
- **The fragile base class problem**
- **Inflexible hierarchy problem** (eventually, all evolving hierarchies are wrong for new uses)
- **The duplication by necessity problem** (due to inflexible hierarchies, new use cases are often shoe-horned in by duplicating, rather than adapting existing code)
- **The Gorilla/banana problem** (What you wanted was a banana, but what you got was a gorilla holding the banana, and the entire jungle)

I discuss some of the issues in more depth in my talk, "Classical Inheritance is Obsolete: How to Think in Prototypal OO":

The solution to all of these problems is to favor object composition over class inheritance.

> *"Favor object composition over class inheritance."*
> *~ The Gang of Four, "Design Patterns: Elements of Reusable Object Oriented Software"*

Summed up nicely here:

Is All Inheritance Bad?

When people say “favor composition over inheritance” that is short for “favor composition over **class** inheritance” (the original quote from “Design Patterns” by the Gang of Four). This is common knowledge in OO design because **class inheritance has many flaws** and causes many problems. Often people leave off the word **class** when they talk about class inheritance, which makes it sound like *all inheritance* is bad —but it’s not.

There are actually several different kinds of inheritance, and most of them are great.

Three Different Kinds of Prototypal Inheritance

Before we dive into the other kinds of inheritance, let’s take a closer look at what I mean by **class inheritance**:

```
// Class Inheritance Example
// NOT RECOMMENDED. Use object composition, instead.

// https://gist.github.com/ericelliott/b668ce0ad1ab540df915
// http://codepen.io/ericelliott/pen/pgdPOb?editors=001

class GuitarAmp {
  constructor ({ cabinet = 'spruce', distortion = '1', volu
    Object.assign(this, {
      cabinet, distortion, volume
    });
  }
}

class BassAmp extends GuitarAmp {
  constructor (options = {}) {
    super(options);
    this.lowCut = options.lowCut;
  }
}

class ChannelStrip extends BassAmp {
  constructor (options = {}) {
    super(options);
    this.inputLevel = options.inputLevel;
  }
}

test('Class Inheritance', nest => {
  nest.test('BassAmp', assert => {
    const msg = `instance should inherit props
    from GuitarAmp and BassAmp`;

```

You can experiment with this example on Codepen.

`*BassAmp*` inherits from `*GuitarAmp*`, and `*ChannelStrip*` inherits from `*BassAmp*` & `*GuitarAmp*`. This is an example of how OO design goes wrong. A channel strip isn't actually a type of guitar amp, and doesn't actually need a cabinet at all. A better option would be to create a new base class that both the amps and the channel strip inherits from, but even that has limitations.

Eventually, the new shared base class strategy breaks down, too.

There’s a better way. You can inherit just the stuff you really need using object composition:

```javascript
// Composition Example

// http://codepen.io/ericelliott/pen/XXzadQ?editors=001
// https://gist.github.com/ericelliott/fed0fd7a0d3388b06402

const distortion = { distortion: 1 };
const volume = { volume: 1 };
const cabinet = { cabinet: 'maple' };
const lowCut = { lowCut: 1 };
const inputLevel = { inputLevel: 1 };

const GuitarAmp = (options) => {
  return Object.assign({}, distortion, volume, cabinet, opt
};

const BassAmp = (options) => {
  return Object.assign({}, lowCut, volume, cabinet, options
};

const ChannelStrip = (options) => {
  return Object.assign({}, inputLevel, lowCut, volume, opti
};


test('GuitarAmp', assert => {
  const msg = 'should have distortion, volume, and cabinet'
  const level = 2;
  const cabinet = 'vintage';

  const actual = GuitarAmp({
    distortion: level,
    volume: level,
    cabinet
  });
  const expected = {
    distortion: level,
    volume: level,
    cabinet
  };

  assert.deepEqual(actual, expected, msg);
  assert.end()
  assert.end();
});

test('BassAmp', assert => {
  const msg = 'should have volume, lowCut, and cabinet';
  const level = 2;
  const cabinet = 'vintage';

  const actual = BassAmp({
    lowCut: level,
    volume: level,
```

Experiment with this on CodePen.

If you look carefully, you might see that we're being much more specific about which objects get which properties because with composition, ***we can***. It wasn't really an option with class inheritance. When you inherit from a class, you get everything, *even if you don't want it.*

At this point, you may be thinking to yourself, "that's nice, but where are the prototypes?"

To understand that, you have to understand that there are three different kinds of prototypal OO.

Concatenative inheritance: The process of inheriting features directly from one object to another by copying the source objects properties. In JavaScript, source prototypes are commonly referred to as **mixins.** Since ES6, this feature has a convenience utility in JavaScript called `*Object.assign()*`. Prior to ES6, this was commonly done with Underscore/Lodash's `*.extend()*` jQuery's `*$.extend()*`, and so on… The composition example above uses concatenative inheritance.

Prototype delegation: In JavaScript, an object may have a link to a prototype for **delegation**. If a property is not found on the object, the lookup is **delegated** to the **delegate prototype,** which may have a link to its own delegate prototype, and so on up the chain until you arrive at `*Object.prototype*`, which is the root delegate. This is the prototype that gets hooked up when you attach to a `*Constructor.prototype*` and instantiate with `*new*`. You can also use `*Object.create()*` for this purpose, and even mix this technique with concatenation in order to flatten multiple prototypes to a single delegate, or extend the object instance after creation.

Functional inheritance: In JavaScript, any function can create an object. When that function is not a constructor (or `*class*`), it's called a **factory function**. Functional inheritance works by producing an object from a factory, and extending the produced object by assigning properties to it directly (using concatenative inheritance). Douglas Crockford coined the term, but functional inheritance has been in common use in JavaScript for a long time.

As you're probably starting to realize, **concatenative inheritance is the secret sauce that enables object composition in JavaScript**, which makes both prototype delegation and functional inheritance a lot more interesting.

When most people think of prototypal OO in JavaScript, *they think of prototype delegation.* By now you should see that they're missing out on a lot. Delegate prototypes aren't the great alternative to class inheritance—**object composition is**.

Why Composition is Immune to the Fragile Base Class Problem

To understand the fragile base class problem and why it doesn't apply to composition, first you have to understand how it happens:

1. `*A*` is the base class
2. `*B*` inherits from `*A*`
3. `*C*` inherits from `*B*`
4. `*D*` inherits from `*B*`

`*C*` calls `*super*`, which runs code in `*B*`. `*B*` calls `*super*` which runs code in `*A*`.

`*A*` and `*B*` contain unrelated features needed by both `*C*` & `*D*`. `*D*` is a new use case, and needs *slightly different* behavior in `*A*`'s init code than `*C*` needs. So the newbie dev goes and tweaks `*A*`'s init code. **`*C*` breaks because it depends on the existing behavior**, and `*D*` starts working.

What we have here are features spread out between *`A`* and *`B`* that *`C`* and *`D`* need to use in various ways. *`C`* and *`D`* don't use every feature of *`A`* and *`B`*… they just want to inherit some stuff that's already defined in *`A`* and *`B`*. But by inheriting and calling *`super`*, **you don't get to be selective about what you inherit**. You inherit everything:

> *"…the problem with object-oriented languages is they've got all this implicit environment that they carry around with them.* ***You wanted a banana but what you got was a gorilla holding the banana*** *and the entire jungle." ~ Joe Armstrong—"Coders at Work"*

With Composition
Imagine you have features instead of classes:

```
feat1, feat2, feat3, feat4
```

`C` needs *`feat1`* and *`feat3`*, *`D`* needs *`feat1`, `feat2`, `feat4`*:

```
const C = compose(feat1, feat3);
const D = compose(feat1, feat2, feat4);
```

Now, imagine you discover that *`D`* needs **slightly different** behavior from *`feat1`*. It doesn't actually need to change *`feat1`*, instead, **you can make a customized version of *`feat1`*** and use that, instead. You can still inherit the existing behaviors from *`feat2`* and *`feat4`* with no changes:

```
const D = compose(custom1, feat2, feat4);
```

And ***`C`*** **remains unaffected**.

The reason this is not possible with class inheritance is because **when you use class inheritance, you buy into the whole existing class**

taxonomy.

If you want to adapt a little for a new use-case, you either end up duplicating parts of the existing taxonomy (the duplication by necessity problem), or you refactor everything that depends on the existing taxonomy to adapt the taxonomy to the new use case due to **the fragile base class problem**.

Composition is immune to both.

You Think You Know Prototypes, but…

If you were taught to build classes or constructor functions and inherit from those, what you were taught was **not prototypal inheritance**. You were taught how to **mimic class inheritance using prototypes**. See "Common Misconceptions About Inheritance in JavaScript".

In JavaScript, class inheritance piggybacks on top of the very rich, flexible prototypal inheritance features built into the language a long time ago, but when you use class inheritance—even the ES6+ `*class*` inheritance built on top of prototypes, you're not using the full power & flexibility of prototypal OO. In fact, you're painting yourself into corners and **opting into all of the class inheritance problems**.

> *Using class inheritance in JavaScript is like driving your new Tesla Model S to the dealer and trading it in for a rusted out 1983 Ford Pinto.*

Stamps: Composable Factory Functions

Most of the time, composition is achieved using factory functions: functions which exist to create object instances. What if there was a standard that makes factory functions composable? There is. It's called The Stamp Specification.

Explore the Series

- What is a Closure?
- What is the Difference Between Class and Prototypal Inheritance?
- What is a Pure Function?

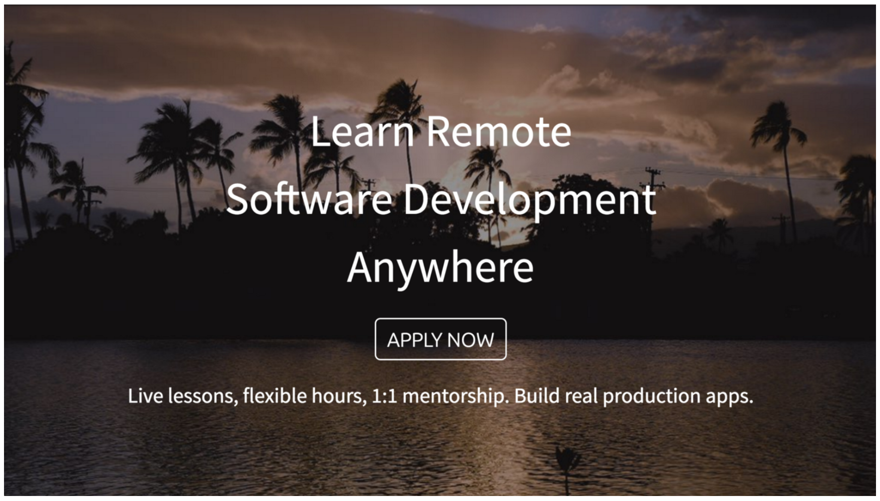
- What is Function Composition?
- What is Functional Programming?
- What is a Promise?
- Soft Skills

. . .

Level Up Your Skills with Live 1:1 Mentorship

DevAnywhere is the fastest way to level up to advanced JavaScript skills:

- Live lessons
- Flexible hours
- 1:1 mentorship
- Build real production apps


Learn Remote
Software Development
Anywhere
APPLY NOW
Live lessons, flexible hours, 1:1 mentorship. Build real production apps.

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.