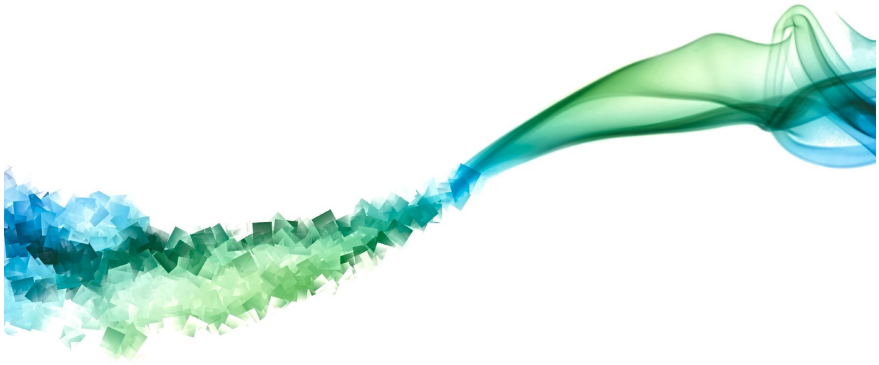
Eric Elliott Follow
Compassionate entrepreneur on a mission to end homelessness.
May 17, 2017 · 11 min read

Composing Software: An Introduction



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

> *Note: This is the introduction to the "Composing Software" series 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!*
> Next >

> *Composition: "The act of combining parts or elements to form a whole." ~ Dictionary.com*

In my first high school programming class, I was told that software development is "the act of breaking a complex problem down into smaller problems, and composing simple solutions to form a complete solution to the complex problem."

One of my biggest regrets in life is that I failed to understand the significance of that lesson early on. I learned the essence of software design far too late in life.

I have interviewed hundreds of developers. What I've learned from those sessions is that I'm not alone. Very few working software developers have a good grasp on the essence of software development.

They aren't aware of the most important tools we have at our disposal, or how to put them to good use. 100% have struggled to answer one or both of the most important questions in the field of software development:

- What is function composition?
- What is object composition?

The problem is that you can't avoid composition just because you're not aware of it. You still do it—but you do it badly. You write code with more bugs, and make it harder for other developers to understand. This is a big problem. The effects are very costly. We spend more time maintaining software than we do creating it from scratch, and our bugs impact billions of people all over the world.

The entire world runs on software today. Every new car is a mini super-computer on wheels, and problems with software design cause real accidents and cost real human lives. In 2013, a jury found Toyota's software development team guilty of "reckless disregard" after an accident investigation revealed spaghetti code with 10,000 global variables.

Hackers and governments stockpile bugs in order to spy on people, steal credit cards, harness computing resources to launch Distributed Denial of Service (DDoS) attacks, crack passwords, and even manipulate elections.

We must do better.

You Compose Software Every Day

If you're a software developer, you compose functions and data structures every day, whether you know it or not. You can do it consciously (and better), or you can do it accidentally, with duct-tape and crazy glue.

The process of software development is breaking down large problems into smaller problems, building components that solve those smaller problems, then composing those components together to form a complete application.

Composing Functions

Function composition is the process of applying a function to the output of another function. In algebra, given two functions, `f` and `g`, `(f ∘ g)(x) = f(g(x))`. The circle is the composition operator. It's commonly pronounced "composed with" or "after". You can say that out-loud as "`f` *composed with* `g` equals `f` of `g` of `x`", or "`f` *after* `g` equals `f` of `g` of `x`". We say `f` *after* `g` because `g` is evaluated first, then its output is passed as an argument to `f`.

Every time you write code like this, you're composing functions:

```
const g = n => n + 1;
const f = n => n * 2;
```

```
const doStuff = x => {
  const afterG = g(x);
  const afterF = f(afterG);
  return afterF;
};
```

```
doStuff(20); // 42
```

Every time you write a promise chain, you're composing functions:

```
const g = n => n + 1;
const f = n => n * 2;
```

```
const wait = time => new Promise(
  (resolve, reject) => setTimeout(
    resolve,
    time
  )
);
```

```
wait(300)
  .then(() => 20)
  .then(g)
  .then(f)
  .then(value => console.log(value)) // 42
;
```

Likewise, every time you chain array method calls, lodash methods, observables (RxJS, etc…) you're composing functions. If you're chaining, you're composing. If you're passing return values into other functions, you're composing. If you call two methods in a sequence, you're composing using `this` as input data.

> *If you're chaining, you're composing.*

When you compose functions intentionally, you'll do it better.

Composing functions intentionally, we can improve our `doStuff()` function to a simple one-liner:

```
const g = n => n + 1;
const f = n => n * 2;

const doStuffBetter = x => f(g(x));

doStuffBetter(20); // 42
```

A common objection to this form is that it's harder to debug. For example, how would we write this using function composition?

```
const doStuff = x => {
  const afterG = g(x);
  console.log(`after g: ${ afterG }`);
  const afterF = f(afterG);
  console.log(`after f: ${ afterF }`);
  return afterF;
};

doStuff(20); // =>
/*
"after g: 21"
"after f: 42"
*/
```

First, let's abstract that "after f", "after g" logging into a little utility called `trace()`:

```
const trace = label => value => {
  console.log(`${ label }: ${ value }`);
  return value;
};
```

Now we can use it like this:

```
const doStuff = x => {
  const afterG = g(x);
  trace('after g')(afterG);
  const afterF = f(afterG);
  trace('after f')(afterF);
  return afterF;
};
```

```
doStuff(20); // =>
/*
"after g: 21"
"after f: 42"
*/
```

Popular functional programming libraries like Lodash and Ramda include utilities to make function composition easier. You can rewrite the above function like this:

```
import pipe from 'lodash/fp/flow';
```

```
const doStuffBetter = pipe(
  g,
  trace('after g'),
  f,
  trace('after f')
);
```

```
doStuffBetter(20); // =>
/*
"after g: 21"
"after f: 42"
*/
```

If you want to try this code without importing something, you can define pipe like this:

```
// pipe(...fns: [...Function]) => x => y
const pipe = (...fns) => x => fns.reduce((y, f) => f(y), x);
```

Don’t worry if you’re not following how that works, yet. Later on we’ll explore function composition in a lot more detail. In fact, it’s so essential, you’ll see it defined and demonstrated many times throughout this text. The point is to help you become so familiar with it that its definition and usage becomes automatic. Be one with the composition.

`pipe()` creates a pipeline of functions, passing the output of one function to the input of another. When you use `pipe()` (and its twin, `compose()`) You don't need intermediary variables. Writing functions without mention of the arguments is called **point-free style.** To do it, you'll call a function that returns the new function, rather than declaring the function explicitly. That means you won't need the `function` keyword or the arrow syntax (`=>`).

Point-free style can be taken too far, but a little bit here and there is great because those intermediary variables add unnecessary complexity to your functions.

There are several benefits to reduced complexity:

Working Memory

The average human brain has only a few shared resources for discrete quanta in working memory, and each variable potentially consumes one of those quanta. As you add more variables, our ability to accurately recall the meaning of each variable is diminished. Working memory models typically involve 4–7 discrete quanta. Above those numbers, error rates dramatically increase.

Using the pipe form, we eliminated 3 variables—freeing up almost half of our available working memory for other things. That reduces our cognitive load significantly. Software developers tend to be better at chunking data into working memory than the average person, but not so much more as to weaken the importance of conservation.

Signal to Noise Ratio

Concise code also improves the signal-to-noise ratio of your code. It's like listening to a radio—when the radio is not tuned properly to the station, you get a lot of interfering noise, and it's harder to hear the music. When you tune it to the correct station, the noise goes away, and you get a stronger musical signal.

Code is the same way. More concise code expression leads to enhanced comprehension. Some code gives us useful information, and some code just takes up space. If you can reduce the amount of code you use without reducing the meaning that gets transmitted, you'll make the code easier to parse and understand for other people who need to read it.

Surface Area for Bugs

Take a look at the before and after functions. It looks like the function went on a diet and lost a ton of weight. That's important because extra code means extra surface area for bugs to hide in, which means more bugs will hide in it.

> *Less code = less surface area for bugs = fewer bugs.*

Composing Objects

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

> *"In computer science, a composite data type or compound data type is any data type which can be constructed in a program using the programming language's primitive data types and other composite types. […] The act of constructing a composite type is known as composition." ~ Wikipedia*

These are primitives:

```
const firstName = 'Claude';
const lastName = 'Debussy';
```

And this is a composite:

```
const fullName = {
  firstName,
  lastName
};
```

Likewise, all Arrays, Sets, Maps, WeakMaps, TypedArrays, etc… are composite datatypes. Any time you build any non-primitive data structure, you're performing some kind of object composition.

Note that the Gang of Four defines a pattern called the **composite pattern** which is a specific type of recursive object composition which allows you to treat individual components and aggregated composites identically. Some developers get confused, thinking that the composite pattern is *the only form of object composition.* Don't get confused. There are many different kinds of object composition.

The Gang of Four continues, "you'll see object composition applied again and again in design patterns", and then they catalog three kinds of object compositional relationships, including **delegation** (as used in the state, strategy, and visitor patterns), **acquaintance** (when an object knows about another object by reference, usually passed as a parameter: a uses-a relationship, e.g., a network request handler might be passed a reference to a logger to log the request—the request handler *uses* a logger), and **aggregation** (when child objects form part of a parent object: a has-a relationship, e.g., DOM children are component elements in a DOM node—A DOM node *has* children).

Class inheritance can be used to construct composite objects, but it's a restrictive and brittle way to do it. When the Gang of Four says "favor object composition over class inheritance", they're advising you to use flexible approaches to composite object building, rather than the rigid, tightly-coupled approach of class inheritance.

We'll use a more general definition of object composition from "Categorical Methods in Computer Science: With Aspects from Topology" (1989):

> *"Composite objects are formed by putting objects together such that each of the latter is 'part of' the former."*

Another good reference is “Reliable Software Through Composite Design”, Glenford J Myers, 1975. Both books are long out of print, but you can still find sellers on Amazon or eBay if you’d like to explore the subject of object composition in more technical depth.

Class inheritance is just one kind of composite object construction. All classes produce composite objects, but not all composite objects are produced by classes or class inheritance. “Favor object composition over class inheritance” means that you should form composite objects from small component parts, rather than inheriting all properties from an ancestor in a class hierarchy. The latter causes a large variety of well-known problems in object oriented design:

- **The tight coupling problem:** Because child classes are dependent on the implementation of the parent class, class inheritance is the tightest coupling available in object oriented design.
- **The fragile base class problem:** Due to tight coupling, changes to the base class can potentially break a large number of descendant classes—potentially in code managed by third parties. The author could break code they’re not aware of.
- **The inflexible hierarchy problem:** With single ancestor taxonomies, given enough time and evolution, all class taxonomies are eventually wrong for new use-cases.
- **The duplication by necessity problem:** Due to inflexible hierarchies, new use cases are often implemented by duplication, rather than extension, leading to similar classes which are unexpectedly divergent. Once duplication sets in, it’s not obvious which class new classes should descend from, or why.
- **The gorilla/banana problem:** “…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”

The most common form of object composition in JavaScript is known as **object concatenation** (aka mixin composition). It works like ice-cream. You start with an object (like vanilla ice-cream), and then mix in the features you want. Add some nuts, caramel, chocolate swirl, and you wind up with nutty caramel chocolate swirl ice cream.

Building composites with class inheritance:

```
class Foo {
  constructor () {
    this.a = 'a'
  }
}

class Bar extends Foo {
  constructor (options) {
    super(options);
    this.b = 'b'
  }
}

const myBar = new Bar(); // {a: 'a', b: 'b'}
```

Building composites with mixin composition:

```
const a = {
  a: 'a'
};

const b = {
  b: 'b'
};

const c = {...a, ...b}; // {a: 'a', b: 'b'}
```

We'll explore other styles of object composition in more depth later. For now, your understanding should be:

1. There's more than one way to do it.
2. Some ways are better than others.
3. You want to select the simplest, most flexible solution for the task at hand.

Conclusion

This isn't about functional programming (FP) vs object-oriented programming (OOP), or one language vs another. Components can take the form of functions, data structures, classes, etc… Different programming languages tend to afford different atomic elements for components. Java affords classes, Haskell affords functions, etc… But no matter what language and what paradigm you favor, you can't get away from composing functions and data structures. In the end, that's what it all boils down to.

We'll talk a lot about functional programming, because functions are the simplest things to compose in JavaScript, and the functional programming community has invested a lot of time and effort formalizing function composition techniques.

What we won't do is say that functional programming is better than object-oriented programming, or that you must choose one over the other. OOP vs FP is a false dichotomy. Every real Javascript application I've seen in recent years mixes FP and OOP extensively.

We'll use object composition to produce datatypes for functional programming, and functional programming to produce objects for OOP.

No matter how you write software, you should compose it well.

> *The essence of software development is composition.*

A software developer who doesn't understand composition is like a home builder who doesn't know about bolts or nails. Building software without awareness of composition is like a home builder putting walls together with duct tape and crazy glue.

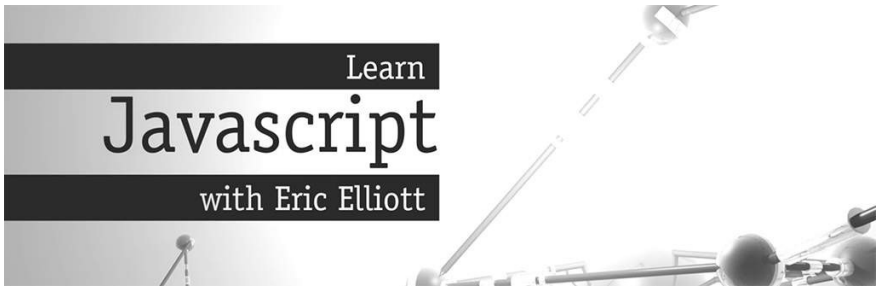
It's time to simplify, and the best way to simplify is to get to the essence. The trouble is, almost nobody in the industry has a good handle on the essentials. We as an industry have failed you, the software developer. It's our responsibility as an industry to train developers better. We must improve. We need to take responsibility. Everything runs on software today, from the economy to medical equipment. There is literally no corner of human life on this planet that is not impacted by the quality of our software. We need to know what we're doing.

It's time to learn how to compose software.

Continued in "The Rise and Fall and Rise of Functional Programming"

Learn More at EricElliottJS.com

Video lessons on function & object composition are available for members of EricElliottJS.com. If you're not a member, sign up today.



. . .

Eric Elliott *is the author of "Programming JavaScript Applications" (O'Reilly), and "Learn JavaScript with Eric Elliott". 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 remote from anywhere with the most beautiful woman in the world.