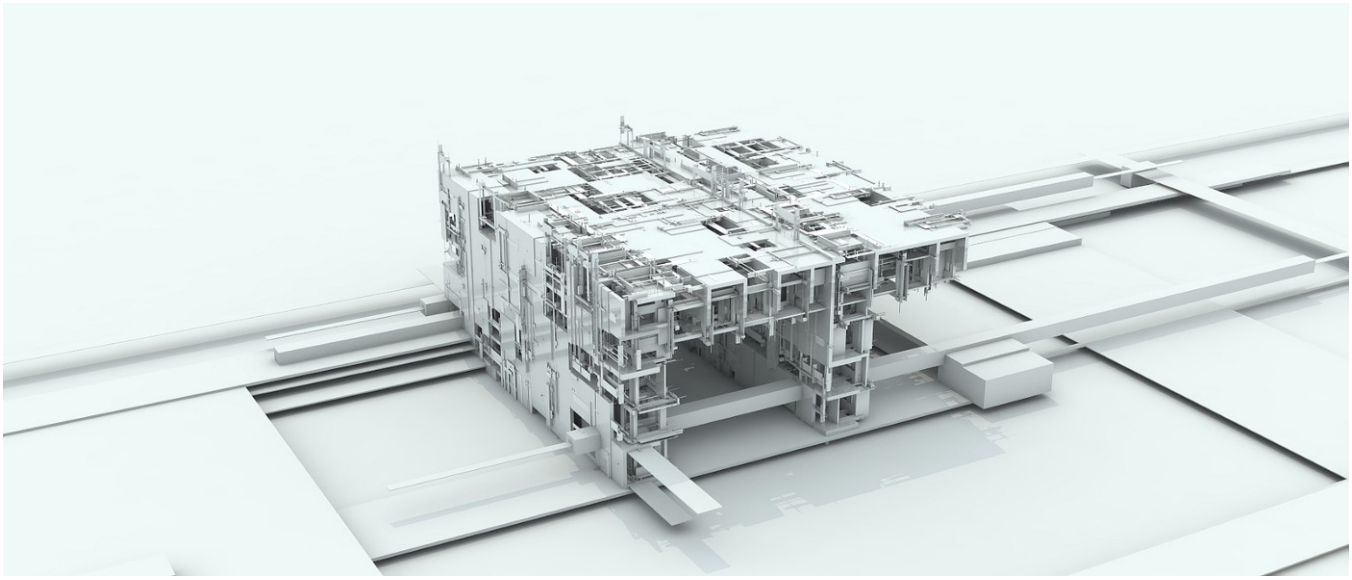
Master the JavaScript Interview: What is Functional Programming?

Eric Elliott
Jan 4, 2017 · 11 min read



Structure Synth — Orihaus (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.*

Functional programming has become a really hot topic in the JavaScript world. Just a few years ago, few JavaScript programmers even knew what functional programming is, but every large application codebase I've seen in the past 3 years makes heavy use of functional programming ideas.

Functional programming (often abbreviated FP) is the process of building software by composing **pure functions**, avoiding **shared state, mutable data,** and **side-effects**. Functional programming is **declarative** rather than **imperative**, and application state

flows through pure functions. Contrast with object oriented programming, where application state is usually shared and colocated with methods in objects.

Functional programming is a **programming paradigm**, meaning that it is a way of thinking about software construction based on some fundamental, defining principles (listed above). Other examples of programming paradigms include object oriented programming and procedural programming.

Functional code tends to be more concise, more predictable, and easier to test than imperative or object oriented code — but if you're unfamiliar with it and the common patterns associated with it, functional code can also seem a lot more dense, and the related literature can be impenetrable to newcomers.

If you start googling functional programming terms, you're going to quickly hit a brick wall of academic lingo that can be very intimidating for beginners. To say it has a learning curve is a serious understatement. But if you've been programming in JavaScript for a while, chances are good that you've used a lot of functional programming concepts & utilities in your real software.

> Don't let all the new words scare you away. It's a lot easier than it sounds.

The hardest part is wrapping your head around all the unfamiliar vocabulary. There are a lot of ideas in the innocent looking definition above which all need to be understood before you can begin to grasp the meaning of functional programming:

- Pure functions
- Function composition
- Avoid shared state
- Avoid mutating state
- Avoid side effects

In other words, if you want to know what functional programming means in practice, you have to start with an understanding of those core concepts.

A **pure function** is a function which:

- Given the same inputs, always returns the same output, and
- Has no side-effects

Pure functions have lots of properties that are important in functional programming, including **referential transparency** (you can replace a function call with its resulting value without changing the meaning of the program). Read “What is a Pure Function?” for more details.

Function composition is the process of combining two or more functions in order to produce a new function or perform some computation. For example, the composition `f . g` (the dot means “composed with”) is equivalent to `f(g(x))` in JavaScript. Understanding function composition is an important step towards understanding how software is constructed using the functional programming. Read “What is Function Composition?” for more.

Shared State

Shared state is any variable, object, or memory space that exists in a shared scope, or as the property of an object being passed between scopes. A shared scope can include global scope or closure scopes. Often, in object oriented programming, objects are shared between scopes by adding properties to other objects.

For example, a computer game might have a master game object, with characters and game items stored as properties owned by that object. Functional programming avoids shared state — instead relying on immutable data structures and pure calculations to derive new data from existing data. For more details on how functional software might handle application state, see “10 Tips for Better Redux Architecture”.

The problem with shared state is that in order to understand the effects of a function, you have to know the entire history of every shared variable that the function uses or affects.

Imagine you have a user object which needs saving. Your `saveUser()` function makes a request to an API on the server. While that’s happening, the user changes their profile picture with `updateAvatar()` and triggers another `saveUser()` request. On save, the server sends back a canonical user object that should replace whatever is in memory in

order to sync up with changes that happen on the server or in response to other API calls.

Unfortunately, the second response gets received before the first response, so when the first (now outdated) response gets returned, the new profile pic gets wiped out in memory and replaced with the old one. This is an example of a race condition — a very common bug associated with shared state.

Another common problem associated with shared state is that changing the order in which functions are called can cause a cascade of failures because functions which act on shared state are timing dependent:

```
// With shared state, the order in which function calls are made
// changes the result of the function calls.
const x = {
  val: 2
};

const x1 = () => x.val += 1;

const x2 = () => x.val *= 2;

x1();
x2();

console.log(x.val); // 6

// This example is exactly equivalent to the above, except...
const y = {
  val: 2
};

const y1 = () => y.val += 1;

const y2 = () => y.val *= 2;

// ...the order of the function calls is reversed...
y2();
y1();

// ... which changes the resulting value:
console.log(y.val); // 5
```

timing-dependency.js hosted with ❤️ by GitHub view raw

Timing dependency example

When you avoid shared state, the timing and order of function calls don't change the result of calling the function. With pure functions, given the same input, you'll always get the same output. This makes function calls completely independent of other function calls, which can radically simplify changes and refactoring. A change in one function, or the timing of a function call won't ripple out and break other parts of the program.

```
const x = {
  val: 2
};

const x1 = x => Object.assign({}, x, { val: x.val + 1});

const x2 = x => Object.assign({}, x, { val: x.val * 2});

console.log(x1(x2(x)).val); // 5


const y = {
  val: 2
};

// Since there are no dependencies on outside variables,
// we don't need different functions to operate on different
// variables.

// this space intentionally left blank


// Because the functions don't mutate, you can call these
// functions as many times as you want, in any order,
// without changing the result of other function calls.
x2(y);
x1(y);

console.log(x1(x2(y)).val); // 5
```

no-timing-dependency.js hosted with ❤ by GitHub view raw

In the example above, we use `Object.assign()` and pass in an empty object as the first parameter to copy the properties of `x` instead of mutating it in place. In this case, it would have been equivalent to simply create a new object from scratch, without

`Object.assign()`, but this is a common pattern in JavaScript to create copies of existing state instead of using mutations, which we demonstrated in the first example.

If you look closely at the `console.log()` statements in this example, you should notice something I've mentioned already: function composition. Recall from earlier, function composition looks like this: `f(g(x))`. In this case, we replace `f()` and `g()` with `x1()` and `x2()` for the composition: `x1 . x2`.

Of course, if you change the order of the composition, the output will change. Order of operations still matters. `f(g(x))` is not always equal to `g(f(x))`, but what doesn't matter anymore is what happens to variables outside the function — and that's a big deal. With impure functions, it's impossible to fully understand what a function does unless you know the entire history of every variable that the function uses or affects.

Remove function call timing dependency, and you eliminate an entire class of potential bugs.

Immutability

An **immutable** object is an object that can't be modified after it's created. Conversely, a **mutable** object is any object which can be modified after it's created.

Immutability is a central concept of functional programming because without it, the data flow in your program is lossy. State history is abandoned, and strange bugs can creep into your software. For more on the significance of immutability, see "The Dao of Immutability."

In JavaScript, it's important not to confuse `const`, with immutability. `const` creates a variable name binding which can't be reassigned after creation. `const` does not create immutable objects. You can't change the object that the binding refers to, but you can still change the properties of the object, which means that bindings created with `const` are mutable, not immutable.

Immutable objects can't be changed at all. You can make a value truly immutable by deep freezing the object. JavaScript has a method that freezes an object one-level deep:

```
1  const a = Object.freeze({
2    foo: 'Hello',
3    bar: 'world',
4    baz: '!'
5  });
```

```
6
7  a.foo = 'Goodbye';
8  // Error: Cannot assign to read only property 'foo' of object Object
```

frozen-objects.js hosted with ❤️ by GitHub view raw

But frozen objects are only superficially immutable. For example, the following object is mutable:

```
1  const a = Object.freeze({
2    foo: { greeting: 'Hello' },
3    bar: 'world',
4    baz: '!'
5  });
6
7  a.foo.greeting = 'Goodbye';
8
9  console.log(`${ a.foo.greeting }, ${ a.bar }${a.baz}`);
```

frozen-not-immutable.js hosted with ❤️ by GitHub view raw

As you can see, the top level primitive properties of a frozen object can't change, but any property which is also an object (including arrays, etc…) can still be mutated — so even frozen objects are not immutable unless you walk the whole object tree and freeze every object property.

In many functional programming languages, there are special immutable data structures called **trie data structures** (pronounced "tree") which are effectively deep frozen — meaning that no property can change, regardless of the level of the property in the object hierarchy.

Tries use **structural sharing** to share reference memory locations for all the parts of the object which are unchanged after an object has been copied by an operator, which uses less memory, and enables significant performance improvements for some kinds of operations.

For example, you can use identity comparisons at the root of an object tree for comparisons. If the identity is the same, you don't have to walk the whole tree checking for differences.

There are several libraries in JavaScript which take advantage of tries, including Immutable.js and Mori.

I have experimented with both, and tend to use Immutable.js in large projects that require significant amounts of immutable state. For more on that, see “10 Tips for Better Redux Architecture”.

Side Effects

A side effect is any application state change that is observable outside the called function other than its return value. Side effects include:

- Modifying any external variable or object property (e.g., a global variable, or a variable in the parent function scope chain)
- Logging to the console
- Writing to the screen
- Writing to a file
- Writing to the network
- Triggering any external process
- Calling any other functions with side-effects

Side effects are mostly avoided in functional programming, which makes the effects of a program much easier to understand, and much easier to test.

Haskell and other functional languages frequently isolate and encapsulate side effects from pure functions using **monads**. The topic of monads is deep enough to write a book on, so we’ll save that for later.

What you do need to know right now is that side-effect actions need to be isolated from the rest of your software. If you keep your side effects separate from the rest of your program logic, your software will be much easier to extend, refactor, debug, test, and maintain.

This is the reason that most front-end frameworks encourage users to manage state and component rendering in separate, loosely coupled modules.

Reusability Through Higher Order Functions

Functional programming tends to reuse a common set of functional utilities to process data. Object oriented programming tends to colocate methods and data in objects. Those colocated methods can only operate on the type of data they were designed to operate on, and often only the data contained in that specific object instance.

In functional programming, any type of data is fair game. The same `map()` utility can map over objects, strings, numbers, or any other data type because it takes a function as an argument which appropriately handles the given data type. FP pulls off its generic utility trickery using **higher order functions**.

JavaScript has **first class functions**, which allows us to treat functions as data — assign them to variables, pass them to other functions, return them from functions, etc…

A **higher order function** is any function which takes a function as an argument, returns a function, or both. Higher order functions are often used to:

- Abstract or isolate actions, effects, or async flow control using callback functions, promises, monads, etc…
- Create utilities which can act on a wide variety of data types
- Partially apply a function to its arguments or create a curried function for the purpose of reuse or function composition
- Take a list of functions and return some composition of those input functions

Containers, Functors, Lists, and Streams

A functor is something that can be mapped over. In other words, it's a container which has an interface which can be used to apply a function to the values inside it. When you see the word functor, you should think "mappable".

Earlier we learned that the same `map()` utility can act on a variety of data types. It does that by lifting the mapping operation to work with a functor API. The important flow control operations used by `map()` take advantage of that interface. In the case of `Array.prototype.map()`, the container is an array, but other data structures can be functors, too — as long as they supply the mapping API.

Let's look at how `Array.prototype.map()` allows you to abstract the data type from the mapping utility to make `map()` usable with any data type. We'll create a simple

`double()` mapping that simply multiplies any passed in values by 2:

```
const double = n => n * 2;
const doubleMap = numbers => numbers.map(double);
console.log(doubleMap([2, 3, 4])); // [ 4, 6, 8 ]
```

double-mapping.js hosted with ❤️ by GitHub view raw

What if we want to operate on targets in a game to double the number of points they award? All we have to do is make a subtle change to the `double()` function that we pass into `map()`, and everything still works:

```
const double = n => n.points * 2;

const doubleMap = numbers => numbers.map(double);

console.log(doubleMap([
  { name: 'ball', points: 2 },
  { name: 'coin', points: 3 },
  { name: 'candy', points: 4}
])); // [ 4, 6, 8 ]
```

map-custom-data-type.js hosted with ❤️ by GitHub view raw

The concept of using abstractions like functors & higher order functions in order to use generic utility functions to manipulate any number of different data types is important in functional programming. You'll see a similar concept applied in all sorts of different ways.

> "A list expressed over time is a stream."

All you need to understand for now is that arrays and functors are not the only way this concept of containers and values in containers applies. For example, an array is just a list of things. A list expressed over time is a stream — so you can apply the same kinds of utilities to process streams of incoming events — something that you'll see a lot when you start building real software with FP.

Declarative vs Imperative

Functional programming is a declarative paradigm, meaning that the program logic is expressed without explicitly describing the flow control.

Imperative programs spend lines of code describing the specific steps used to achieve the desired results — the **flow control: How** to do things.

Declarative programs abstract the flow control process, and instead spend lines of code describing the **data flow: What** to do. The *how* gets abstracted away.

For example, this **imperative** mapping takes an array of numbers and returns a new array with each number multiplied by 2:

```
const doubleMap = numbers => {
  const doubled = [];
  for (let i = 0; i < numbers.length; i++) {
    doubled.push(numbers[i] * 2);
  }
  return doubled;
};

console.log(doubleMap([2, 3, 4])); // [4, 6, 8]
```

imperative-example.js hosted with ❤️ by GitHub — view raw

Imperative data mapping

This **declarative** mapping does the same thing, but abstracts the flow control away using the functional `Array.prototype.map()` utility, which allows you to more clearly express the flow of data:

```
const doubleMap = numbers => numbers.map(n => n * 2);

console.log(doubleMap([2, 3, 4])); // [4, 6, 8]
```

declarative-example.js hosted with ❤️ by GitHub — view raw

Imperative code frequently utilizes statements. A **statement** is a piece of code which performs some action. Examples of commonly used statements include `for`, `if`, `switch`, `throw`, etc...

Declarative code relies more on expressions. An **expression** is a piece of code which evaluates to some value. Expressions are usually some combination of function calls, values, and operators which are evaluated to produce the resulting value.

These are all examples of expressions:

```
2 * 2
doubleMap([2, 3, 4])
Math.max(4, 3, 2)
```

Usually in code, you'll see expressions being assigned to an identifier, returned from functions, or passed into a function. Before being assigned, returned, or passed, the expression is first evaluated, and the resulting value is used.

Conclusion

Functional programming favors:

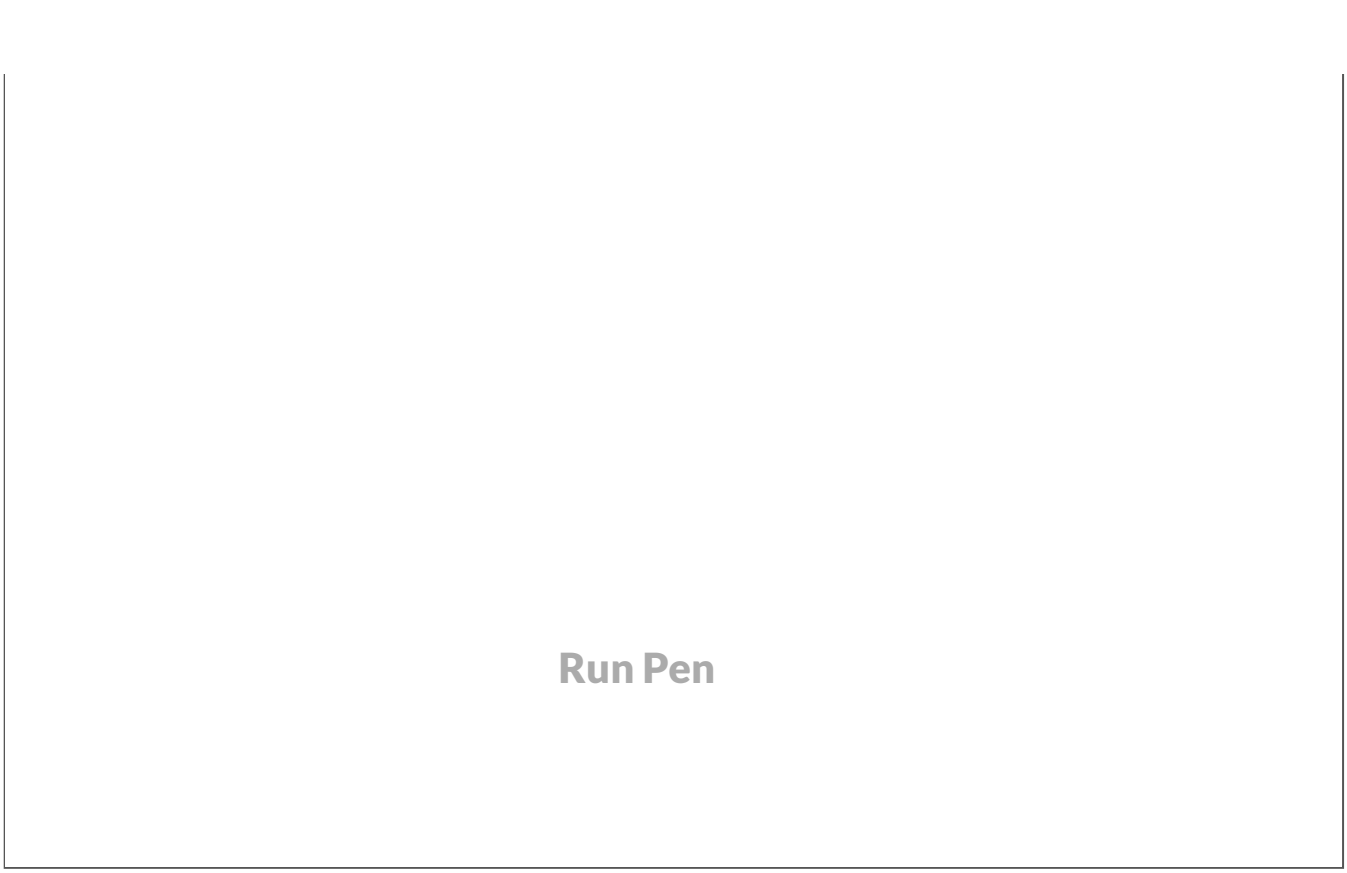
- Pure functions instead of shared state & side effects
- Immutability over mutable data
- Function composition over imperative flow control
- Lots of generic, reusable utilities that use higher order functions to act on many data types instead of methods that only operate on their colocated data
- Declarative rather than imperative code (what to do, rather than how to do it)
- Expressions over statements
- Containers & higher order functions over ad-hoc polymorphism

Homework

Learn & practice this core group of functional array extras:

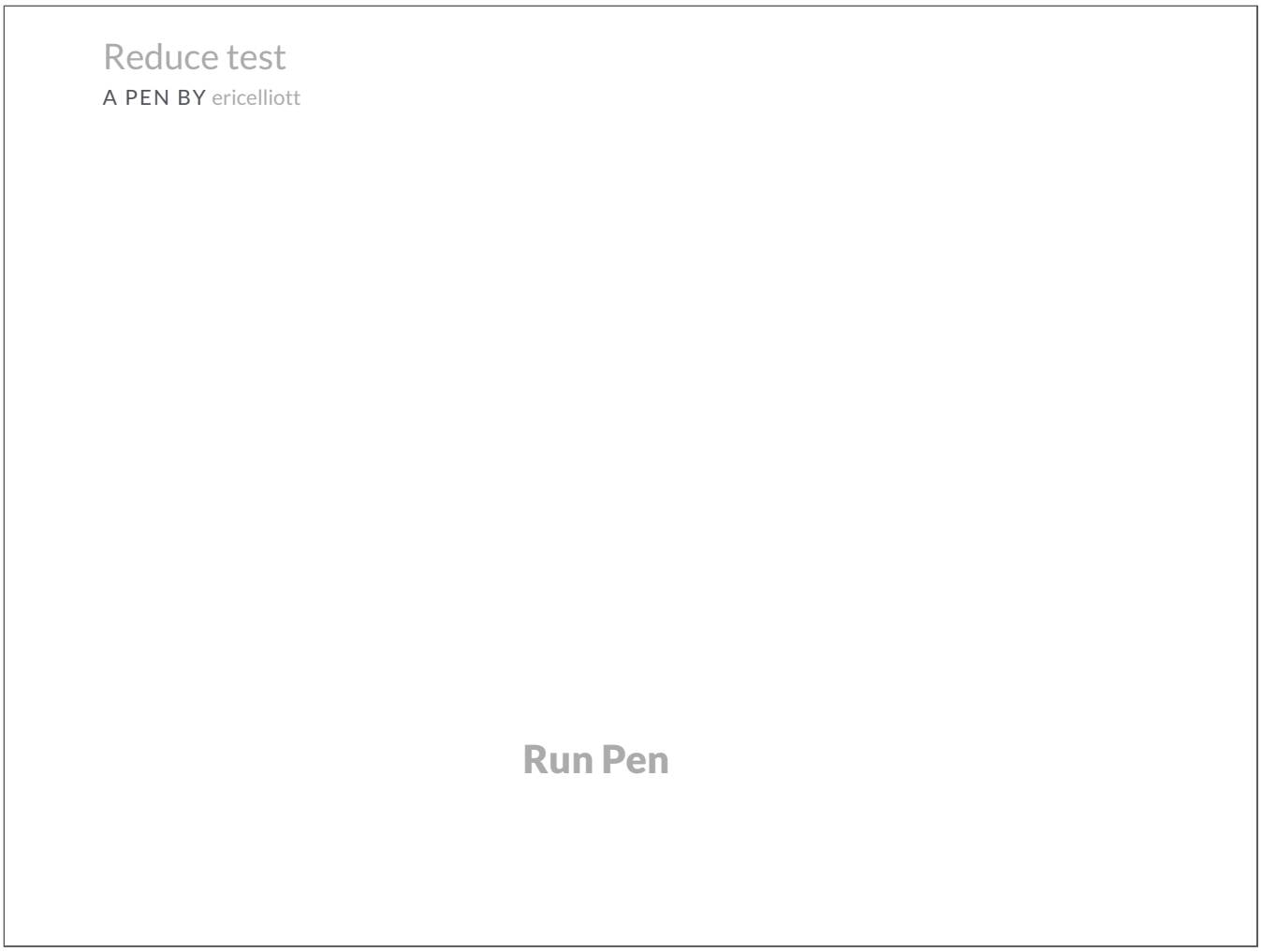
- `.map()`
- `.filter()`
- `.reduce()`

Use map to transform the following array of values into an array of item names:



Use filter to select the items where points are greater than or equal to 3:

Use reduce to sum the points:



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

> *This post was included in the book "Composing Software".*
> Buy the Book | Index | < Previous | Next >

. . .

Start your free lesson on EricElliottJS.com

Eric Elliott *is the author of the books, "Composing Software" and "Programming JavaScript Applications". As co-founder of EricElliottJS.com and DevAnywhere.io, he teaches developers essential software development skills. He builds and advises development teams for crypto projects, and has contributed to software experiences for* ***Adobe Systems, Zumba Fitness, The Wall StreetJournal, 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 Functional Programming Technology