Eric Elliott Follow
Compassionate entrepreneur on a mission to end homelessness.
Mar 25, 2016 · 9 min read

Master the JavaScript Interview: What is a Pure Function?

Image: Pure — carnagenyc (CC-BY-NC 2.0)

Pure functions are essential for a variety of purposes, including functional programming, reliable concurrency, and React+Redux apps. But what does “pure function” mean?

We're going to answer this question with a free lesson from "Learn JavaScript with Eric Elliott":

Before we can tackle what a pure function is, it's probably a good idea to take a closer look at functions. There may be a different way to look at them that will make functional programming easier to understand.

What is a Function?

A **function** is a process which takes some input, called **arguments**, and produces some output called a **return value**. Functions may serve the following purposes:

- **Mapping:** Produce some output based on given inputs. A function **maps** input values to output values.
- **Procedures:** A function may be called to perform a sequence of steps. The sequence is known as a procedure, and programming in this style is known as **procedural programming**.
- **I/O:** Some functions exist to communicate with other parts of the system, such as the screen, storage, system logs, or network.

Mapping

Pure functions are all about mapping. Functions map input arguments to return values, meaning that for each set of inputs, there exists an output. A function will take the inputs and return the corresponding output.

`*Math.max()*` takes numbers as arguments and returns the largest number:

```
Math.max(2, 8, 5); // 8
```

In this example, 2, 8, & 5 are *arguments*. They're values passed into the function.

`*Math.max()*` is a function that takes any number of arguments and returns the largest argument value. In this case, the largest number we passed in was 8, and that's the number that got returned.

Functions are really important in computing and math. They help us process data in useful ways. Good programmers give functions descriptive names so that when we see the code, we can see the function names and understand what the function does.

Math has functions, too, and they work a lot like functions in JavaScript. You've probably seen functions in algebra. They look something like this:

$f(x) = 2x$

Which means that we're declaring a function called f and it takes an argument called x and multiplies x by 2.

To use this function, we simply provide a value for x:

$f(2)$

In algebra, this means exactly the same thing as writing:

4

So any place you see $f(2)$ you can substitute 4.

Now let's convert that function to JavaScript:

```
const double = x => x * 2;
```

You can examine the function's output using *`console.log()`*:

```
console.log( double(5) ); // 10
```

Remember when I said that in math functions, you could replace *`f(2)`* with *`4`*? In this case, the JavaScript engine replaces *`double(5)`* with the answer, *`10`*.

So, *`console.log( double(5) );`* is the same as *`console.log(10);`*

This is true because *`double()`* is a pure function, but if *`double()`* had side-effects, such as saving the value to disk or logging to the console, you couldn't simply replace *`double(5)`* with 10 without changing the meaning.

If you want referential transparency, you need to use pure functions.

Pure Functions

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

- Given the same input, will always return the same output.
- Produces no side effects.

> *A dead giveaway that a function is impure is if it makes sense to call it without using its return value. For pure functions, that's a noop.*

I recommend that you favor pure functions. Meaning, if it is practical to implement a program requirement using pure functions, you should use them over other options. Pure functions take some input and return some output based on that input. They are the simplest reusable building blocks of code in a program. Perhaps the most important design principle in computer science is KISS (Keep It Simple, Stupid). I

prefer Keep It Stupid Simple. Pure functions are stupid simple in the best possible way.

Pure functions have many beneficial properties, and form the foundation of **functional programming**. Pure functions are completely independent of outside state, and as such, they are immune to entire classes of bugs that have to do with shared mutable state. Their independent nature also makes them great candidates for parallel processing across many CPUs, and across entire distributed computing clusters, which makes them essential for many types of scientific and resource-intensive computing tasks.

Pure functions are also extremely independent—easy to move around, refactor, and reorganize in your code, making your programs more flexible and adaptable to future changes.

The Trouble with Shared State

Several years ago I was working on an app that allowed users to search a database for musical artists and load the artist's music playlist into a web player. This was around the time Google Instant landed, which displays instant search results as you type your search query. AJAX-powered autocomplete was suddenly all the rage.

The only problem was that users often type faster than an API autocomplete search response can be returned, which caused some strange bugs. It would trigger race conditions, where newer suggestions would be replaced by outdated suggestions.

Why did that happen? Because each AJAX success handler was given access to directly update the suggestion list that was displayed to users. The slowest AJAX request would always win the user's attention by blindly replacing results, even when those replaced results may have been newer.

To fix the problem, I created a suggestion manager—a single source of truth to manage the state of the query suggestions. It was aware of a currently pending AJAX request, and when the user typed something new, the pending AJAX request would be canceled before a new request was issued, so only a single response handler at a time would ever be able to trigger a UI state update.

Any sort of asynchronous operation or concurrency could cause similar race conditions. Race conditions happen if output is dependent on the sequence of uncontrollable events (such as network, device latency, user input, randomness, etc…). In fact, if you're using shared state and that state is reliant on sequences which vary depending on indeterministic factors, for all intents and purposes, the output is impossible to predict, and that means it's impossible to properly test or fully understand. As Martin Odersky (creator of Scala) puts it:

> non-determinism = parallel processing + mutable state

Program determinism is usually a desirable property in computing. Maybe you think you're OK because JS runs in a single thread, and as such, is immune to parallel processing concerns, but as the AJAX example demonstrates, a single threaded JS engine does not imply that there is no concurrency. On the contrary, there are many sources of concurrency in JavaScript. API I/O, event listeners, web workers, iframes, and timeouts can all introduce indeterminism into your program. Combine that with shared state, and you've got a recipe for bugs.

Pure functions can help you avoid those kinds of bugs.

Given the Same Input, Always Return the Same Output

With our `*double()*` function, you can replace the function call with the result, and the program will mean the same thing— `*double(5)*` will always mean the same thing as `*10*` in your program, regardless of context, no matter how many times you call it or when.

But you can't say the same thing about all functions. Some functions rely on information other than the arguments you pass in to produce results.

Consider this example:

```
Math.random(); // => 0.4011148700956255
Math.random(); // => 0.8533405303023756
Math.random(); // => 0.3550692005082965
```

Even though we didn't pass any arguments into any of the function calls, they all produced different output, meaning that *`Math.random()`* is **not pure**.

`Math.random()` produces a new random number between 0 and 1 every time you run it, so clearly you couldn't just replace it with 0.4011148700956255 without changing the meaning of the program.

That would produce the same result every time. When we ask the computer for a random number, it usually means that we want a different result than we got the last time. What's the point of a pair of dice with the same numbers printed on every side?

Sometimes we have to ask the computer for the current time. We won't go into the details of how the time functions work. For now, just copy this code:

```
const time = () => new Date().toLocaleTimeString();

time(); // => "5:15:45 PM"
```

What would happen if you replaced the *`time()`* function call with the current time?

It would always say it's the same time: the time that the function call got replaced. In other words, it could only produce the correct output once per day, and only if you ran the program at the exact moment that the function got replaced.

So clearly, *`time()`* isn't like our *`double()`* function.

A function is only pure if, given the same input, it will always produce the same output. You may remember this rule from algebra class: the same input values will always map to the same output value. However, many input values may map to the same output value. For example, the following function **is pure**:

```
const highpass = (cutoff, value) => value >= cutoff;
```

The same input values will always map to the same output value:

```
highpass(5, 5); // => true
highpass(5, 5); // => true
highpass(5, 5); // => true
```

Many input values may map to the same output value:

```
highpass(5, 123); // true
highpass(5, 6);   // true
highpass(5, 18);  // true
```

```
highpass(5, 1);   // false
highpass(5, 3);   // false
highpass(5, 4);   // false
```

A pure function must not rely on any external mutable state, because it would no longer be deterministic or referentially transparent.

Pure Functions Produce No Side Effects

A pure function produces no side effects, which means that it can't alter any external state.

Immutability

JavaScript's object arguments are references, which means that if a function were to mutate a property on an object or array parameter, that would mutate state that is accessible outside the function. Pure functions must not mutate external state.

Consider this mutating, **impure** *`addToCart()`* function:

```
// impure addToCart mutates existing cart
const addToCart = (cart, item, quantity) => {
  cart.items.push({
    item,
    quantity
  });
  return cart;
};


test('addToCart()', assert => {
  const msg = 'addToCart() should add a new item to the car
  const originalCart =     {
    items: []
  };
  const cart = addToCart(
    originalCart,
    {
      name: "Digital SLR Camera",
      price: '1495'
    },
```

It works by passing in a cart, and item to add to that cart, and an item quantity. The function then returns the same cart, with the item added to it.

The problem with this is that we've just mutated some shared state. Other functions may be relying on that cart object state to be what it was before the function was called, and now that we've mutated that shared state, we have to worry about what impact it will have on the program logic if we change the order in which functions have been called. Refactoring the code could result in bugs popping up, which could screw up orders, and result in unhappy customers.

Now consider this version:

```
// Pure addToCart() returns a new cart
// It does not mutate the original.
const addToCart = (cart, item, quantity) => {
  const newCart = lodash.cloneDeep(cart);

  newCart.items.push({
    item,
    quantity
  });
  return newCart;

};


test('addToCart()', assert => {
  const msg = 'addToCart() should add a new item to the car
  const originalCart = {
    items: []
  };

  // deep-freeze on npm
  // throws an error if original is mutated
  deepFreeze(originalCart);

  const cart = addToCart(
    originalCart,
    {
      name: "Digital SLR Camera",
```

In this example, we have an array nested in an object, which is why I reached for a deep clone. This is more complex state than you'll typically be dealing with. For most things, you can break it down into smaller chunks.

For example, Redux lets you compose reducers rather than deal with the entire app state inside each reducer. The result is that you don't have to create a deep clone of the entire app state every time you want to update just a small part of it. Instead, you can use non-destructive array methods, or `*Object.assign()*` to update a small part of the app state.

Your turn. Fork this pen and change the impure functions into pure functions. Make the unit tests pass without changing the tests.

Explore the Series

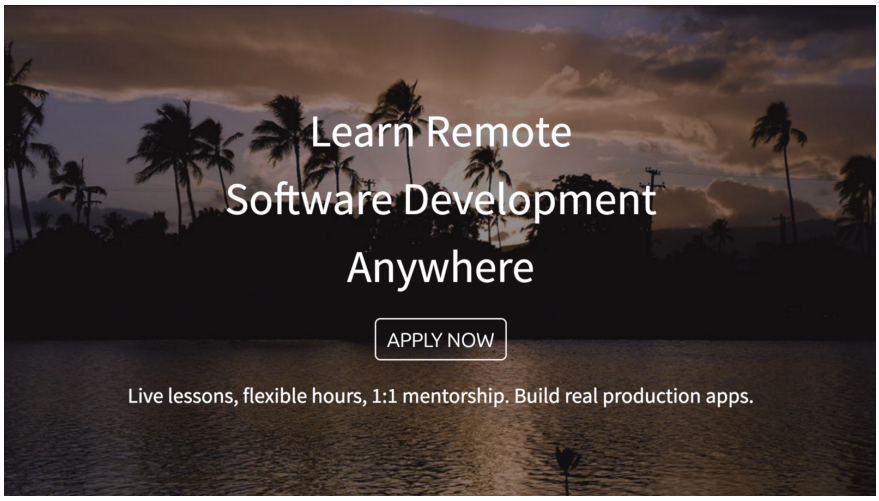
- What is a Closure?
- What is the Difference Between Class and Prototypal Inheritance?
- What is a Pure Function?
- What is Function Composition?
- What is Functional Programming?
- What is a Promise?
- Soft Skills

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



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.