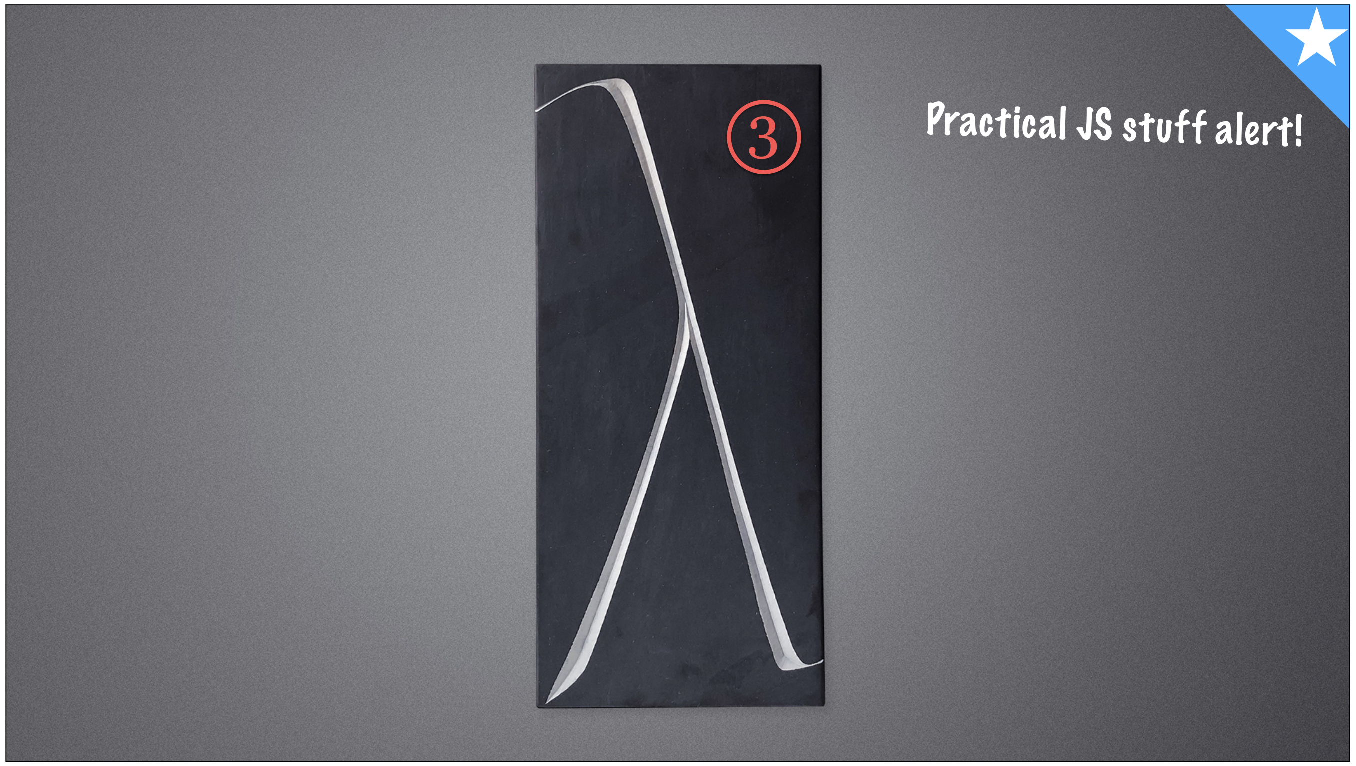


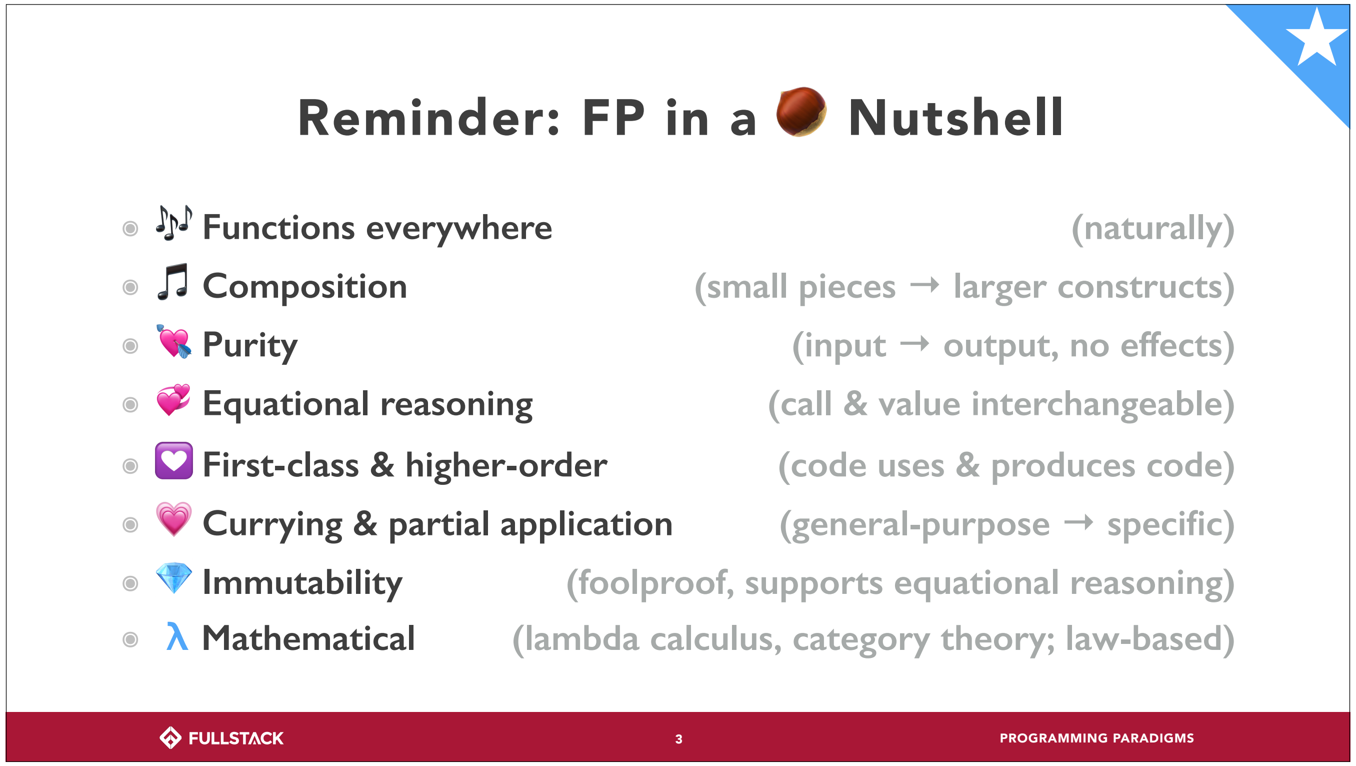
Practical JS stuff alert!

Here comes the practical bit!

FP Fundamentals in JS

Pure Functions · Function Composition · Currying & Partial Application

We are going to see some *fundamental* FP concepts as expressed / commonly used in JS. This won't cover everything, but it'll be a start.



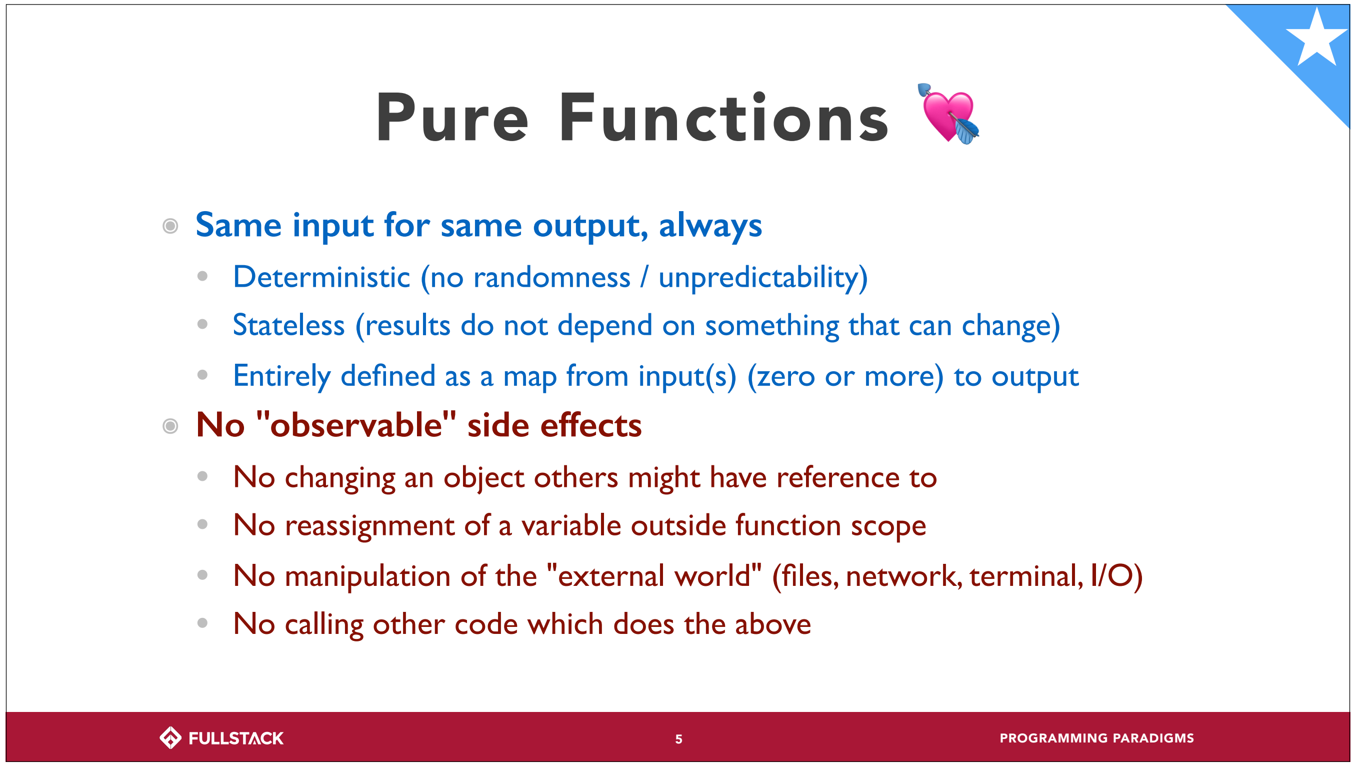
Reminder: FP in a 🌰 Nutshell

- 🎶 Functions everywhere (naturally)
- 🎵 Composition (small pieces → larger constructs)
- 💘 Purity (input → output, no effects)
- 💞 Equational reasoning (call & value interchangeable)
- 💟 First-class & higher-order (code uses & produces code)
- 💗 Currying & partial application (general-purpose → specific)
- 💎 Immutability (foolproof, supports equational reasoning)
- λ Mathematical (lambda calculus, category theory; law-based)

Just a reminder!

Pure 💘 Functions

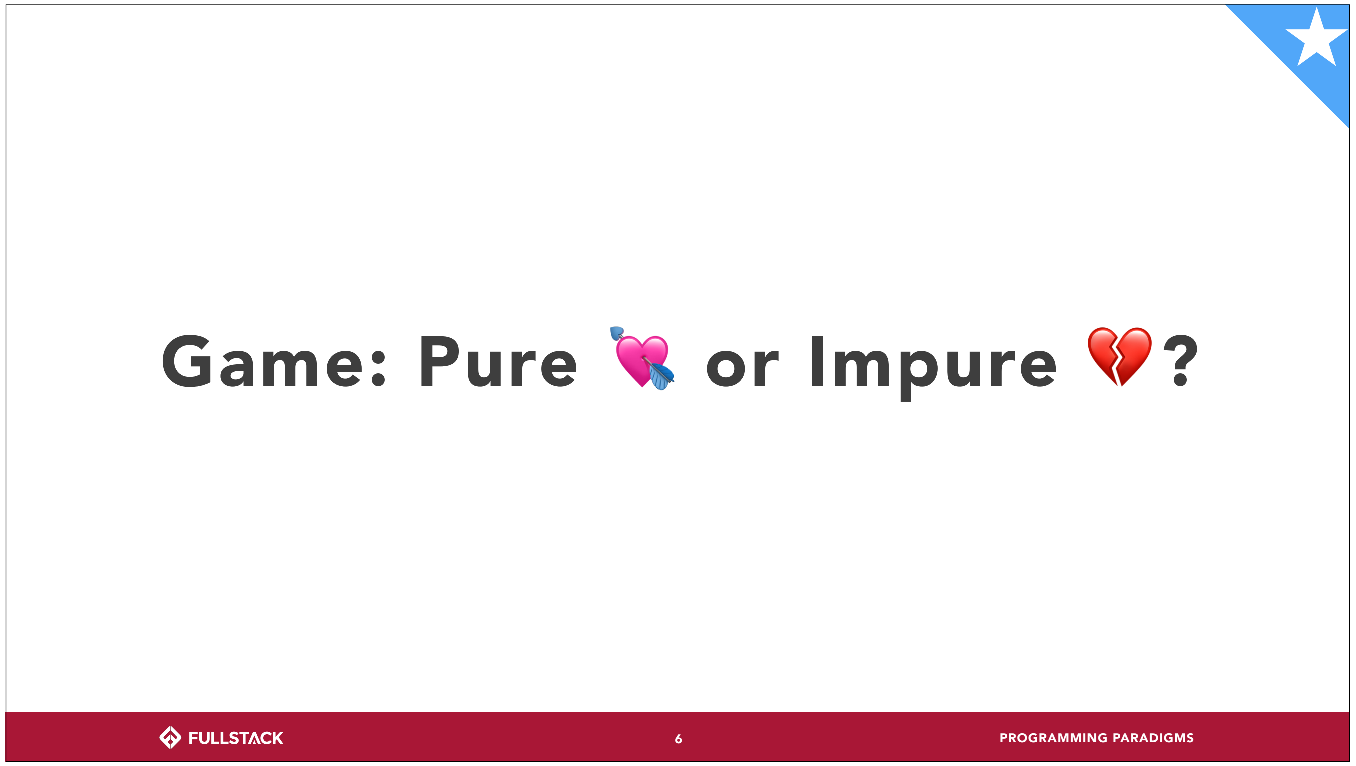
input → output, no side effects



Pure Functions 💘

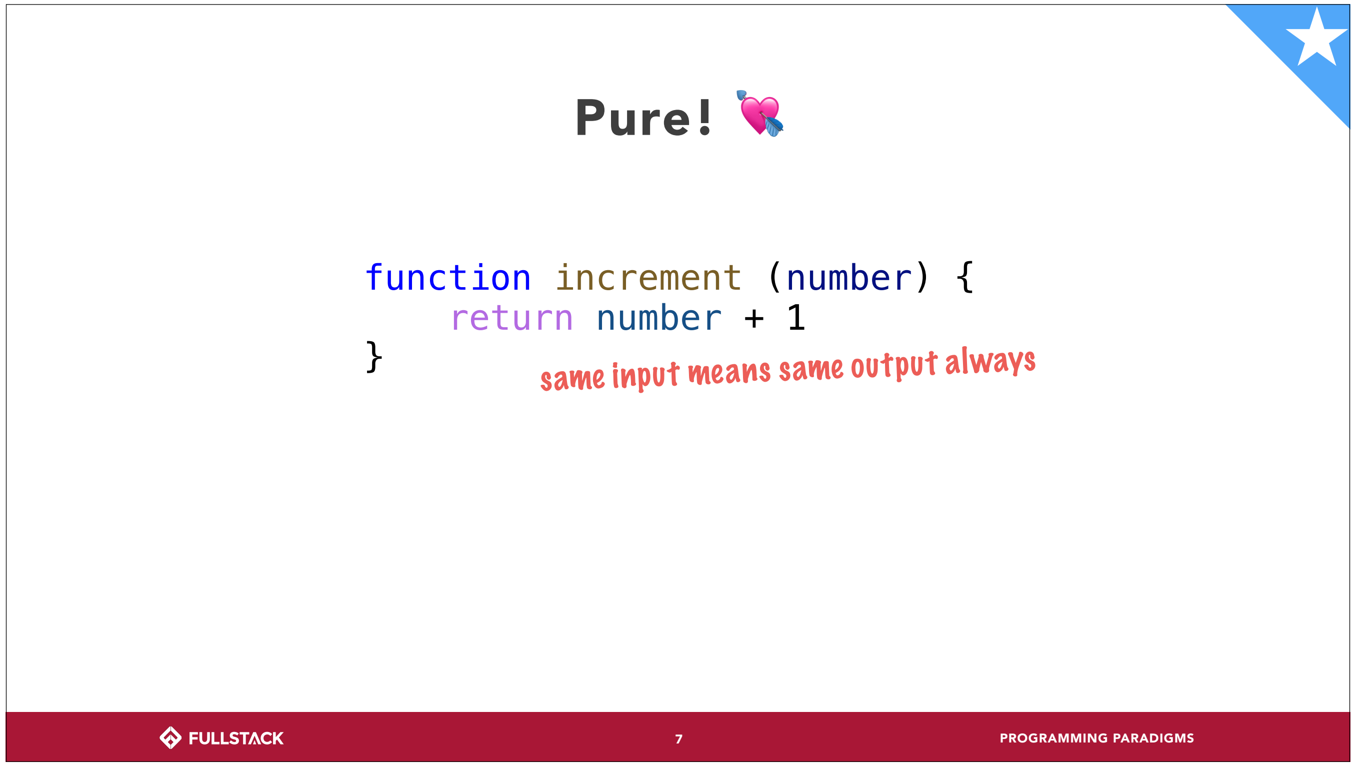
- **Same input for same output, always**
  - Deterministic (no randomness / unpredictability)
  - Stateless (results do not depend on something that can change)
  - Entirely defined as a map from input(s) (zero or more) to output
- **No "observable" side effects**
  - No changing an object others might have reference to
  - No reassignment of a variable outside function scope
  - No manipulation of the "external world" (files, network, terminal, I/O)
  - No calling other code which does the above

This is a dual definition – pure functions are as much about what they *don't* do as what they *do* do.



Game: Pure 💘 or Impure 💔?

Let's play a game! You might have students raise their hands to vote on pure/impure.



Pure! 💘

```
function increment (number) {
    return number + 1
}
```

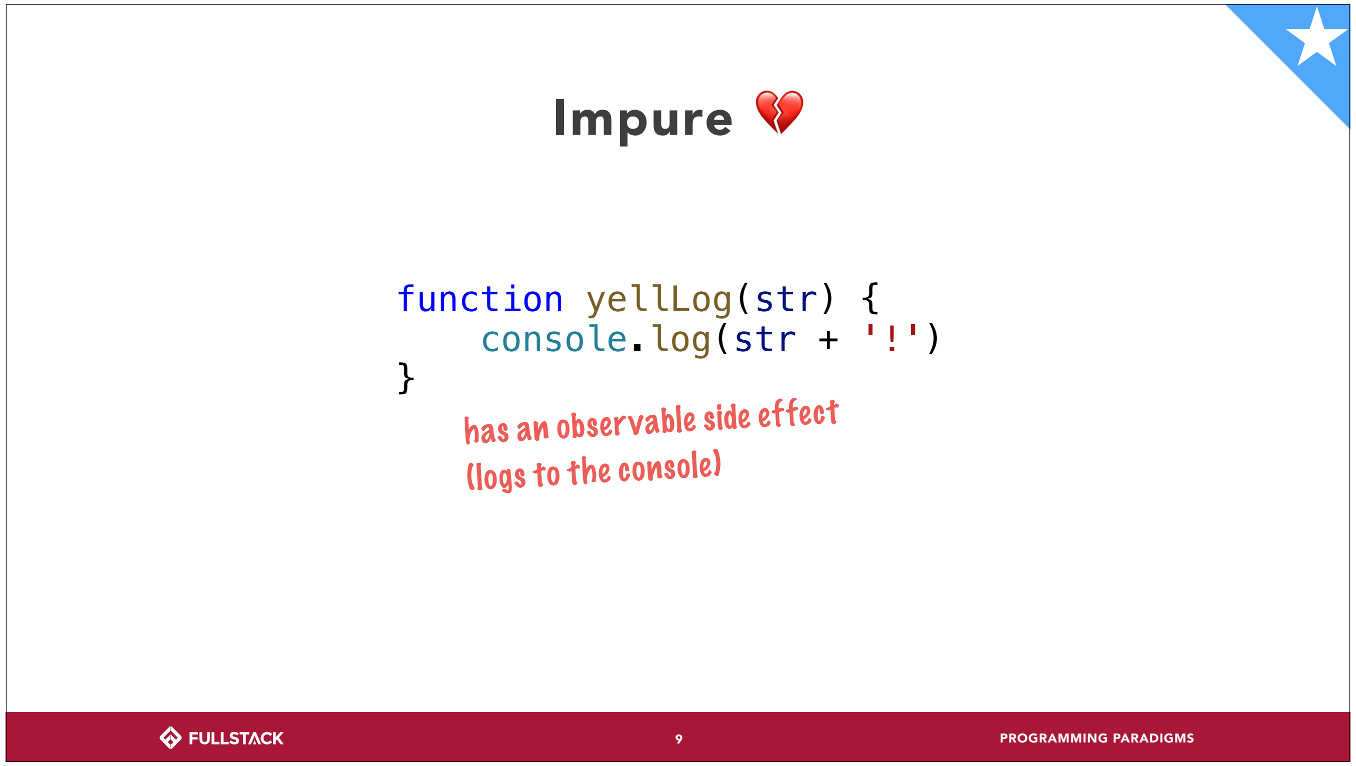
same input means same output always

Easy start.

Impure 💔

```
function grow (person) {
    person.age = person.age + 1
    return person
}
```

mutates object that others
might have or get a reference to

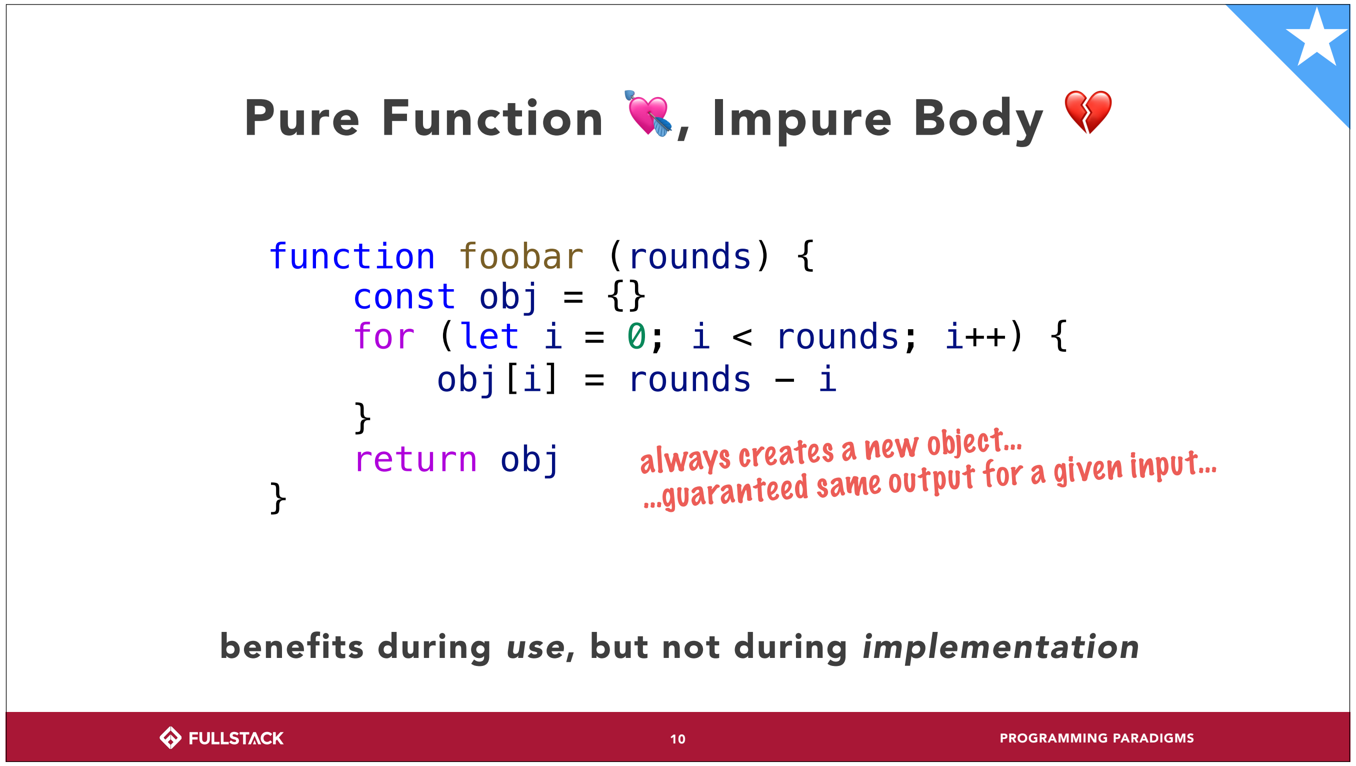


Impure 💔

```
function yellLog(str) {
    console.log(str + '!')
}
```

has an observable side effect
(logs to the console)

Why is logging to the console "bad"? Well, we cannot necessarily replace this function call with its return statement – we need the function to run *at a particular time* during the execution of the code. This breaks equational reasoning – we are back to the imperative idea of *sequencing* steps.



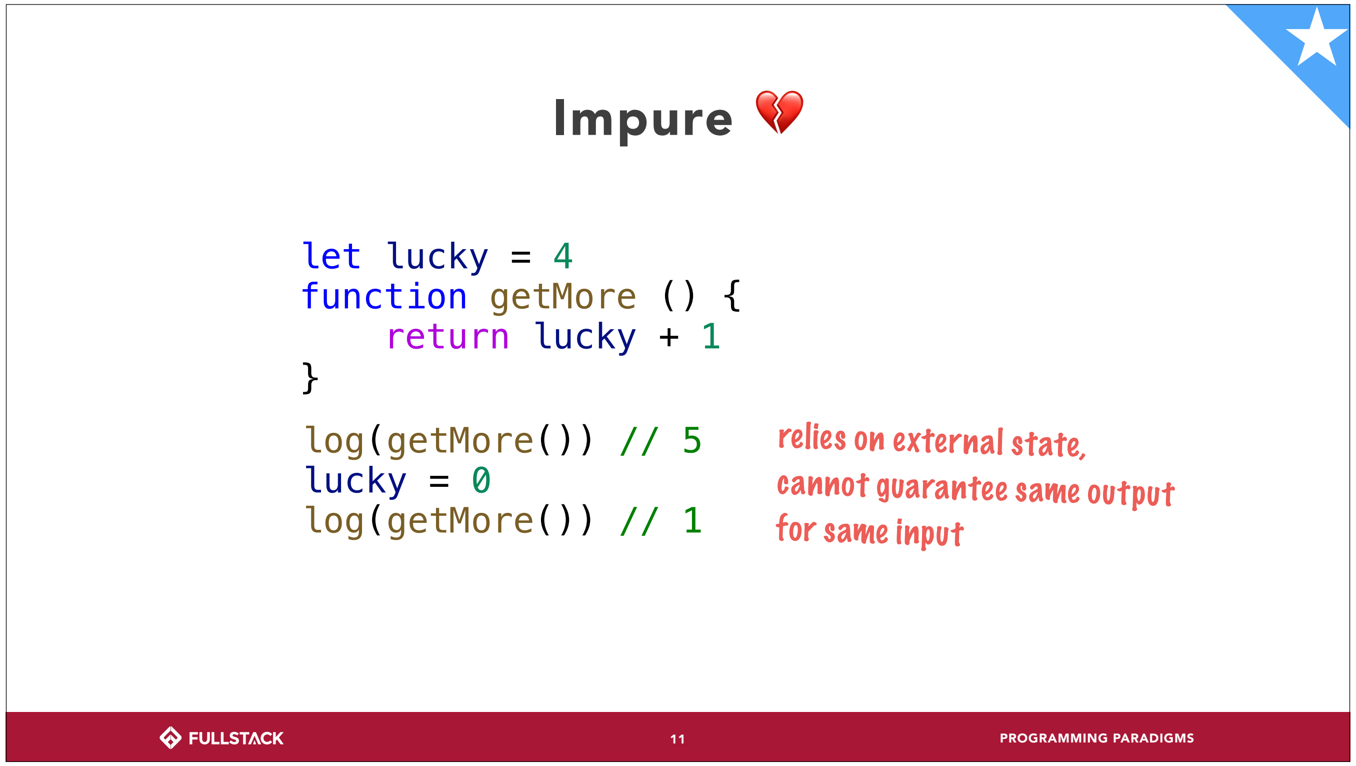
Pure Function 💘, Impure Body 💔

```
function foobar (rounds) {
    const obj = {}
    for (let i = 0; i < rounds; i++) {
        obj[i] = rounds - i
    }
    return obj
}
```

always creates a new object...
...guaranteed same output for a given input...

benefits during *use*, but not during *implementation*

I might also call this one "observably pure" but not "internally pure". If you were given this function as a black box, it would be pure for all intents and purposes. But while you the developer are *writing* this function, you are using impure techniques (granted, ones that do not extend outside of the function body).



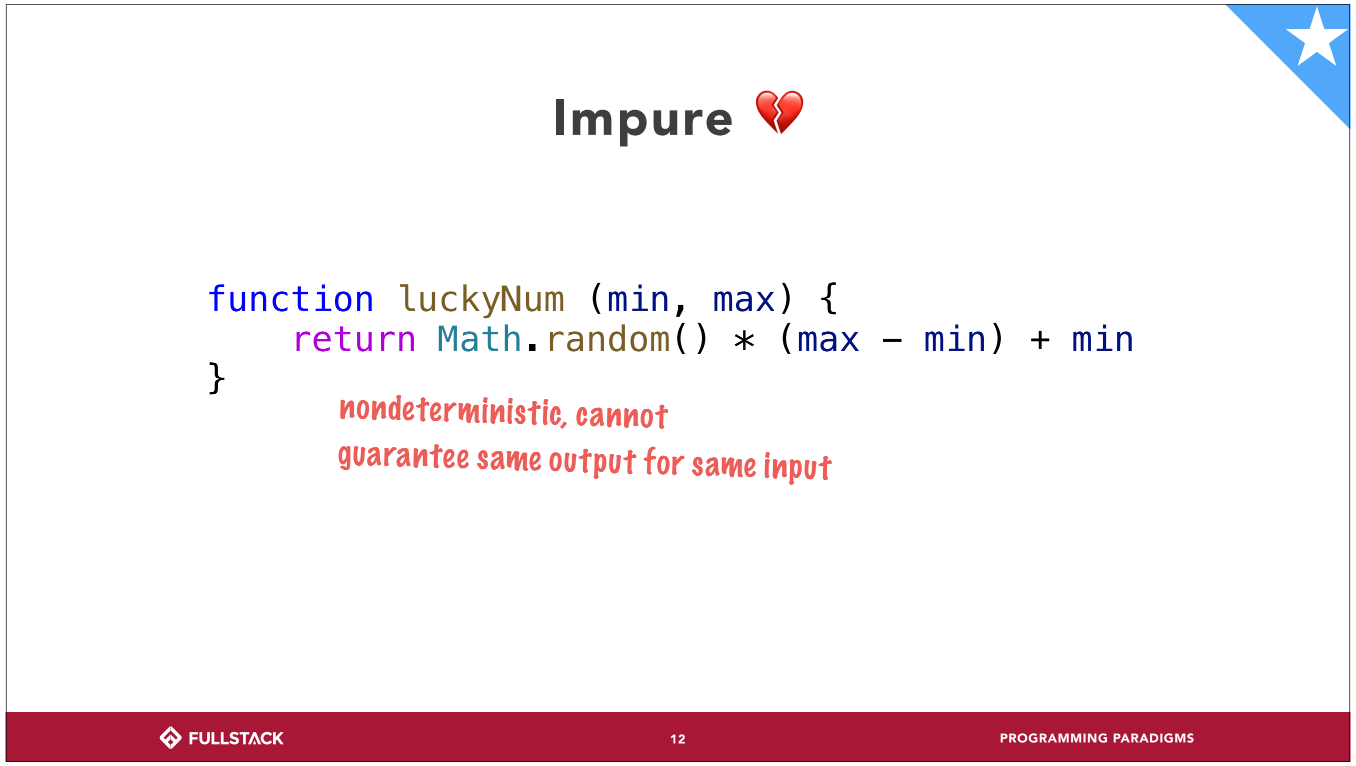
Impure 💔

```
let lucky = 4
function getMore () {
    return lucky + 1
}

log(getMore()) // 5
lucky = 0
log(getMore()) // 1
```

relies on external state,
cannot guarantee same output
for same input

Definitely impure.

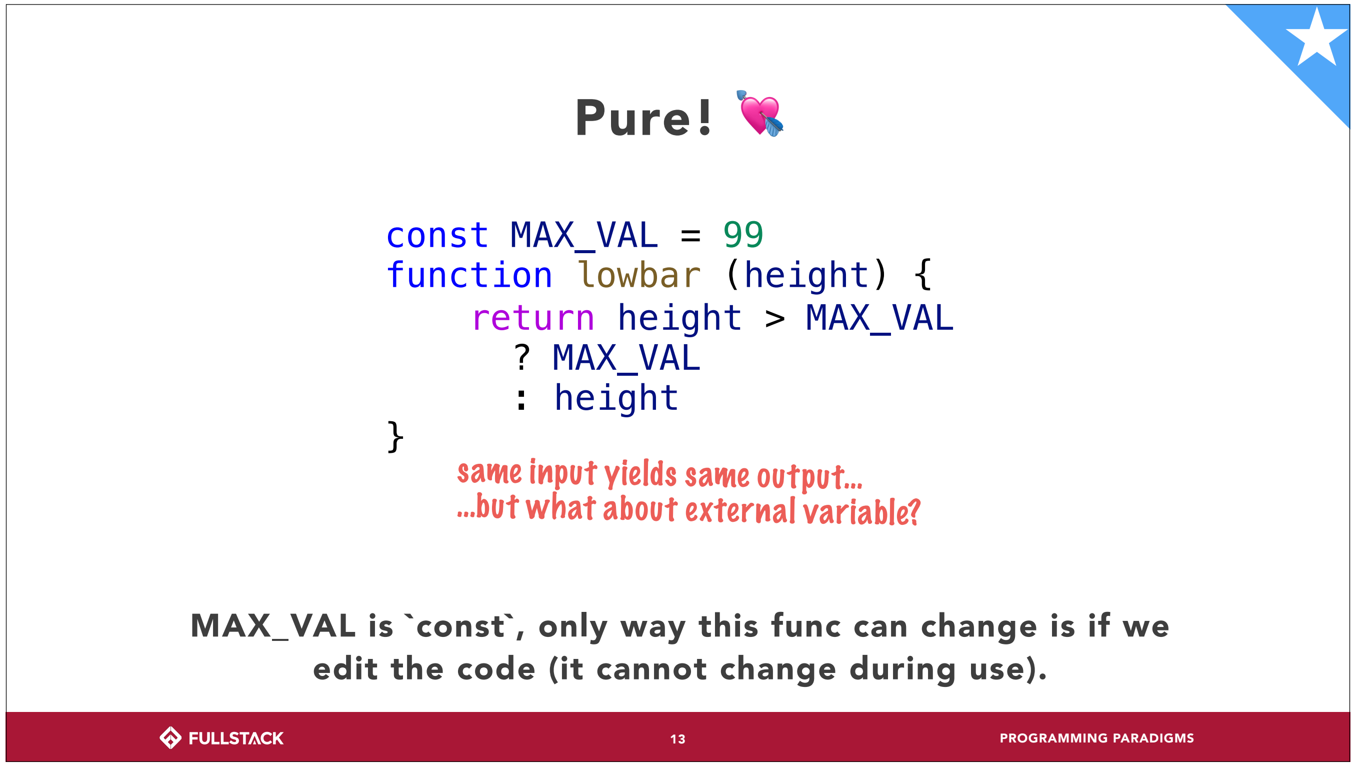


Impure 💔

```
function luckyNum (min, max) {
    return Math.random() * (max - min) + min
}
```

nondeterministic, cannot
guarantee same output for same input

Also quite clearly impure.



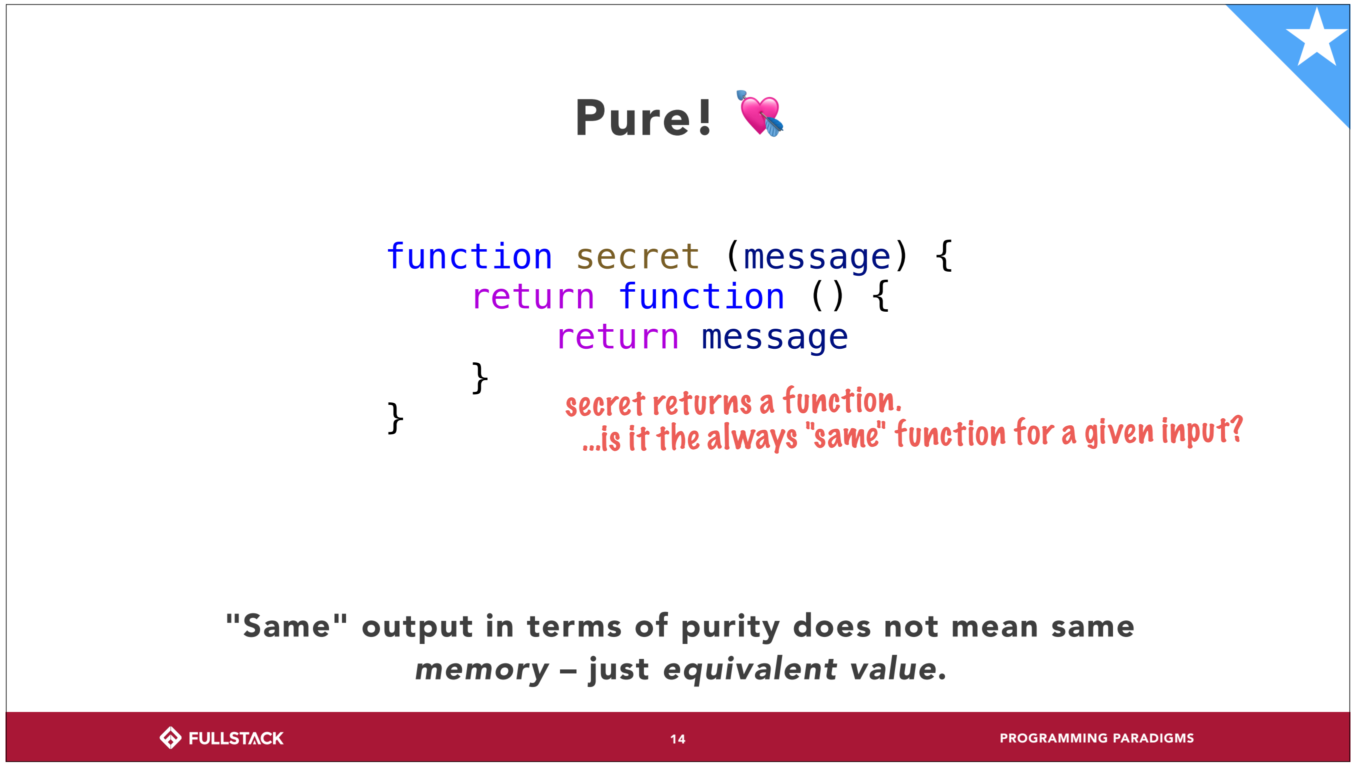
Pure! 💘

```
const MAX_VAL = 99
function lowbar (height) {
    return height > MAX_VAL
      ? MAX_VAL
      : height
}
```

same input yields same output...
...but what about external variable?

MAX_VAL is `const`, only way this func can change is if we edit the code (it cannot change during use).

This isn't necessarily the best way to write this function – it is harder to copy-paste a function which depends on an external variable. But as far as the *runtime program* is concerned, this function is pure; during runtime, it will never do anything but give the same output for the same input, with no side effects.



Pure! 💘

```
function secret (message) {
    return function () {
        return message
    }
}
```

secret returns a function.
...is it the always "same" function for a given input?

"Same" output in terms of purity does not mean same *memory* – just *equivalent value*.

Value equivalency is a big thing in FP. For example, in Haskell, `[1, 2] == [1, 2]`. It doesn't matter where things are stored because you can exchange one `[1, 2]` list for another and your code will behave identically. Only in a language with mutation does it matter if the arrays are the same in memory.

Why?

So why do we like pure functions, anyway?

Pure Functions 💘

- 🧠 **Afford you strong reasoning capabilities**
  - Can move around, invoke anywhere, and nothing will break
  - Do not have to think about how you got to a pure function – *only* inputs and outputs. No need to replay entire program in your head!
- 💯 **Very easy to test**
  - Put stuff in, get something out. If it maps as you intend, it's working.
- 🎵 **Very easy to compose**
  - Glue pure functions to other pure funcs as you wish, they chain together without causing any issues.

WORKSHOP

PairExercise: Jamda (60 minutes before lunch).