✨ A FUNctional JavaScript Makeover ✨

Mike Schutte

Detroit JavaScript Meetup

June 18, 2018

🥅 Goals 🥅

- Identify and use currying and partial application
- Speak to the pros and cons of "functions first, data last"
  - Reconsider how the order of arguments affects functions

λ FP: The Essence λ

1. Break down a problem into a set of problems that are as small and/or simple as possible.
2. Write a FUNction for each small problem.
3. Stitch together those smaller FUNctions via composition to solve the larger problem at hand.

OOP vs FP

- OOP: everything is an object!
  - Collaborating classes/objects via messages
  - *information* first, transformations second
- FP: everything is a function!
  - FUNction composition via type alignment
  - *transformations* first, information second

🎉 Javascript is...both! 🎉

Definitions 📕

Pure FUNctions 🐼

- Same argument(s) === same return value. ALWAYS.
- No side-effects.

```
// pure
const increaseCount = (count, value) => count + value
// impure
let count = 0
const increaseCount = value => (count += value)
```

Side-effects 🦑

- mutating data (opposite of "immutability")
- network requests
- updating state
- File I/O

i.e., essential but sketchy things that can have unpredictable outcomes.

the messy stuff 🙀 is the fun stuff 😸

just isolate it ⚠️

.Methods() vs. FUNctions(f)

- Methods are messages sent to objects
  - `user.buildProfile()`
- Functions process inputs
  - `buildProfile(user)`

Predicate functions

- return true or false

```
users.filter(user => !!user.firstName)
```

Higher order FUNctions 🏔

take one or more FUNctions as arguments

AND/OR

return a FUNction

Array.prototype.map/reduce/filter

all higher order FUNctions.

```
const ages = [10, 72, 90, 44]
ages.map(/* oOo */ age => age % 2 === 0 /* FUNction argument! */)
```

Pointfree style

```
const double = number => number * 2
// ages.map(age => double(age))
ages.map(double)
```

"Points" is a synonym for "arguments". Pointfree syntax omits anonymous functions used to delegate arguments.

🤔 Cryptic at first, but...

- Less cognitive clutter
- It forces us to think more about the transformation being done than about the data being transformed.
- By giving the data a name, we "anchor" our thoughts to that data and restrict our understanding of a FUNction's ability. By leaving the data argument out, we can think in more creative and flexible ways.

Currying 🍛

- Convert multi-argument FUNctions to a series of FUNctions that take one argument each (unary) and return a FUNction that takes the next argument.

```
// from
const add = (x, y) => x + y
// to
const add = x => y => x + y
```

Partial Application 🥝

- A function without all of its required arguments is considered "partially applied"

```
const add = x => y => x + y
// partially applied:
const increment = add(1) // y => 1 + y
increment(10) // => 11 (fully applied)
increment(20) // => 21 (fully applied)
```

📖 📖 📖

```
const books = [
  { title: "To Kill A Mockingbird", author: "Harper Lee", year: 1960 },
  { title: "The Secret History", author: "Donna Tartt", year: 1992 },
  { title: "Infinite Jest", author: "David Foster Wallace", year: 1996 },
  { title: "Fight Club", author: "Chuck Palahniuk", year: 1996 },
]
```

Imperative 🍦

```
const booksInYear = (books, year) => {
  let matches = []
  for (book in books) {
    if (book.year === year) {
      matches.push(book)
    }
  }
  return matches
}

booksInYear(books, 1996)
/* => [
{ title: "Infinite Jest", author: "David Foster Wallace", year: 1996 },
{ title: "Fight Club", author: "Chuck Palahniuk", year: 1996 }
]; */
```

Declarative 🍩

```
/*const booksInYear = (books, year) => {
  let matches = []
  for (book in books) {
    if (book.year === year) {
      matches.push(book)
    }
  }
  return matches
}*/

const booksInYear = (year, books) => books.filter(book => book.year === year)
```

```
/* Full Application -- less reusable */
const booksInYear = (year, books) => books.filter(book => book.year === year)
booksInYear(1996, books)
booksInYear(1996, otherBooks)

/* Partial Application -- more reusable */
const booksInYear = year => books => books.filter(book => book.year === year)
const in96 = booksInYear(1996)
in96(books)
in96(otherBooks)
```

Why attach this to books and years?

```
const booksInYear = year => books => books.filter(book => book.year === year)
```

We are really just looking at equality between a property on an object and a given value.

Books and years are not as reusable and generalizable as objects, properties, and values.

Let's Remove Books! 📚 -> 🗑️

```
// const booksInYear = year => books => books.filter(book => book.year === year);
const filterByYear = year => list => list.filter(item => year === item.year)
const in96 = filterByYear(1996)
const movies = [
  { title: "Fargo", year: 1996 },
  { title: "The Shape of Water", year: 2018 },
]

in96(movies) // => [ { title: "Fargo", year: 1996 } ]

in96(books)
/* => [
  { title: "Infinite Jest", author: "David Foster Wallace", year: 1996 },
  { title: "Fight Club", author: "Chuck Palahniuk", year: 1996 }
]; */
```

FUNctions first, data last...

```
const filterByYear = year => list => list.filter(item => year === item.year)
```

Our generic `list` data is coming last in the argument chain (👍), but then we are calling the `list` first with the `filter` *method*.

Data Last -- Part 1 of 3 DLE

Let's turn the `.filter` *method* into a `filter` *FUNction*:

```
const filter = predicate => filterable => filterable.filter(predicate)
const filterByYear = year => list => filter(item => year === item.year)(list)
```

Now `list` comes last as an invoking argument, so we can change this to pointfree syntax:

```
const filterByYear = year => filter(item => year === item.year)
```

🔮

We've removed references to books, `movies`, and even a general `list`, so we are free to think about `filterByYear` as a general purpose FUNctions that filters something based on a year property.

We are still attached to the anchors of `item` and year, and providing a lot of *how* for our solution.

```
const filterByYear = year => filter(item => year === item.year)
```

Data Last -- Part 2 of 3 🌜

Let's turn the `.year` *property* into a prop *FUNction*:

and let's turn the `===` *operator* into a `equals` *FUNction*:

```
const prop = name => obj => obj[name]
const equals = a => b => a === b
const filterByYear = year => filter(item => equals(year)(prop("year")(item)))
```

That looks disgusting! 🤮

```
year => filter(item => equals(year)(prop("year")(item)))
```

⚖️

```
filter(item => equals(year)(prop("year")(item)))
```

There is a weird tipping point of clarity with FP and pointfree style. To make this all worth it, we need to reach for one more critical tool in our FP toolkit: composition!

Simplest composition

inner to outer

```
const exclaim = str => `${str}!!!`
const toUpper = str => str.toUpperCase()
const repeat = str => `${str} ${str}`
const freakout = str => exclaim(toUpper(repeat(str)))
freakout("hey") // => "HEY HEY!!!"
```

`pipe` 🌊

- `pipe` is a higher order FUNction.
- It takes a list of one or more FUNctions, and returns a FUNction.
- That return FUNction takes one or more arguments.
- Those arguments start the "pipeline", where the output of the FUNction on the left is the input for the FUNction to its right.

```js
const pipe = (...FUNs) => startingValue =>
  FUNs.reduce((returnValue, FUN) => FUN(returnValue), startingValue)

const exclaim = str => `${str}!!!`
const toUpper = str => str.toUpperCase()
const repeat = str => `${str} ${str}`

// const freakout = str => exclaim(toUpper(repeat(str)))
const freakout = pipe(
  repeat,
  toUpper,
  exclaim,
)
freakout("hey") // => "HEY HEY!!!"
```

Naming FUNctions

```
const pipe = (...FUNs) => startingValue =>
  FUNs.reduce((returnValue, FUN) => FUN(returnValue), startingValue)

const split = char => str => str.split(char) // => Array
const reverseArr = arr => arr.reverse() // => Array
const join = char => arr => arr.join(char) // => String

//const reverseStr = str => pipe(split(''), reverse, join(''))(str);
const reverseStr = pipe(
  split(""),
  reverse,
  join(""),
) // pointfree
reverseStr("kayak") // => "kayak"
reverseStr("Javascript") // => "tpircsavaJ"
```

Data Last -- Part 3 of 3 ⛅

```
const prop = name => obj => obj[name]
const equals = a => b => a === b

// FROM
year => filter(item => equals(year)(prop("year")(item)))

// TO
year =>
  filter(item =>
    pipe(
      prop("year"),
      equals(year),
    )(item),
  )
```

Now that `item` is simply an invoked argument on the end the return FUNction from `pipe`, we can convert our `filter` FUNction to be pointfree:

```
// year => filter(item => pipe(prop("year"), equals(year))(item));
year =>
  filter(
    pipe(
      prop("year"),
      equals(year),
    ),
  )
```

To clean this up more, let's pull out `filter`'s argument to a named FUNction.

```
// FROM
year => filter(item => equals(year)(prop("year")(item))) // gross

// TO
const yearEquals = year =>
  pipe(
    prop("year"),
    equals(year),
  )
year => filter(yearEquals(year)) // nice
```

BUT WAIT! 🛑

Do we need year?

```
// FROM
const yearEquals = year =>
  pipe(
    prop("year"),
    equals(year),
  )
// the specifics here are "year" and `year`, let's make those arguments in that order
// TO
const propEquals = name => value =>
  pipe(
    prop(name),
    equals(value),
  )
```

```
// const yearEquals = year => propEquals("year")(year);
// const yearEquals = propEquals("year"); // pointfree
// in this case, the API for propEquals("year") is similar to yearEquals
// so let's skip the const assignment
propEquals("year")
```

...and now we are just reading inner-to-outer, so let's pipe:

```
// FROM
year => filter(propEquals("year"))(year)

// TO
year =>
  pipe(
    propEquals("year"),
    filter,
  )(year)
```

…and now that year comes last, we can go pointfree!

```
// FROM
const filterByYear = year =>
  pipe(
    propEquals("year"),
    filter,
  )(year)

// TO
const filterByYear = pipe(
  propEquals("year"),
  filter,
) // pointfree
```

Review 🤕

```
// beginning
const booksInYear = (books, year) => {
  let matches = []
  for (book in books) {
    if (book.year === year) {
      matches.push(book)
    }
  }
  return matches
}

// middle
const filterByYear = year =>
  filter(item => equals(year)(prop("year")(item)))(year) // wtgdf

// end
const filterByYear = pipe(
  propEquals("year"),
  filter,
)
```

Remove the year anchor ⚓

```
const filterBy = propName =>
  pipe(
    propEquals(propName),
    filter,
  )
filterBy("year")
```

All together 😁

```
// units
const prop = name => obj => obj[name]
const equals = a => b => a === b
const pipe = (...FUNs) => startingValue =>
  FUNs.reduce((returnValue, FUN) => FUN(returnValue), startingValue)
const propEquals = name => value =>
  pipe(
    prop(name),
    equals(value),
  )
const filter = predicate => filterable => filterable.filter(predicate)
```

All together 😁

```
// compositions
const filterBy = propName =>
  pipe(
    propEquals(propName),
    filter,
  )
const in96 = filterBy("year")(1996)

in96(movies) // => [ { title: "Fargo", year: 1996 } ]
in96(books)
/* => [
  { title: "Infinite Jest", author: "David Foster Wallace", year: 1996 },
  { title: "Fight Club", author: "Chuck Palahniuk", year: 1996 }
]; */
```

Now *that* is FUNctional.

```
const prop = name => obj => obj[name] // FUNFUN
const equals = a => b => a === b // FUNFUN
const pipe = (...FUNs) => startingValue =>
  FUNs.reduce((returnValue, FUN) => FUN(returnValue), startingValue) // FUNFUN
const propEquals = name => value =>
  pipe(
    prop(name),
    equals(value),
  ) // FUNFUNFUN
const filter = predicate => filterable => filterable.filter(predicate) // FUNFUN

const filterBy = propName =>
  pipe(
    propEquals(propName),
    filter,
  ) // FUNFUNFUN
const filterByYear = filterBy("year") // FUNFUN
const in96 = filterByYear(1996) // FUN

in96(movies) /* DATA */ // => [ { title: "Fargo", year: 1996 } ]
in96(books) /* DATA */
/* => [
  { title: "Infinite Jest", author: "David Foster Wallace", year: 1996 },
  { title: "Fight Club", author: "Chuck Palahniuk", year: 1996 }
]; */
```

New Product Requirements!

- A list of all the titles needs to be shown on an index page.
- All titles should be lowercase, because it's hip 🎩

No problem! 💪

```
const map = mapper => mappable => mappable.map(mapper)
const toLower = str => str.toLowerCase()
pipe(
  in96,
  map(
    pipe(
      prop("title"),
      toLower,
    ),
  ),
)(books)
/* => [ "infinite jest", "fight club" ]; */
```

No problem! 💪

```
const map = mapper => mappable => mappable.map(mapper)
const toLower = str => str.toLowerCase()
const lowerTitle = pipe(
  prop("title"),
  toLower,
)
pipe(
  in96,
  map(lowerTitle),
)(books)
/* => [ "infinite jest", "fight club" ]; */
```

Tradeoffs ⚖️

```
/* library code (e.g., Ramda) */
const pipe = (...FUNs) => startingValue =>
  FUNs.reduce((returnValue, FUN) => FUN(returnValue), startingValue)
const prop = name => obj => obj[name] // can be used for any object
const equals = a => b => a === b
const propEquals = name => value =>
  pipe(
    prop(name),
    equals(value),
  ) // can be used for any object
const filter = predicate => filterable => filterable.filter(predicate)
/* library code (e.g., Ramda) */
```

Tradeoffs ⚖️

```
// more explicit, easier to read
// reusable for different sets of ([{ year }], year)
const booksInYear = (books, year) => {
  let matches = []
  for (book in books) {
    if (book.year === year) {
      matches.push(book)
    }
  }
  return matches
}

/* vs */

const filterBy = propName =>
  pipe(
    propEquals(propName),
    filter,
  ) // can be used for any prop name
const filterByYear = filterBy("year") // context specific helper
const in96 = filterByYear(1996) // can be reused for anything "year-able" and"filter-able"
```

🥅 Goals 🥅

- Identify and use currying and partial application
- Speak to the pros and cons of "functions first, data last"
  - Reconsider how the order of arguments affects functions

Thank you! 🙏

Have *f*UN out there 😉