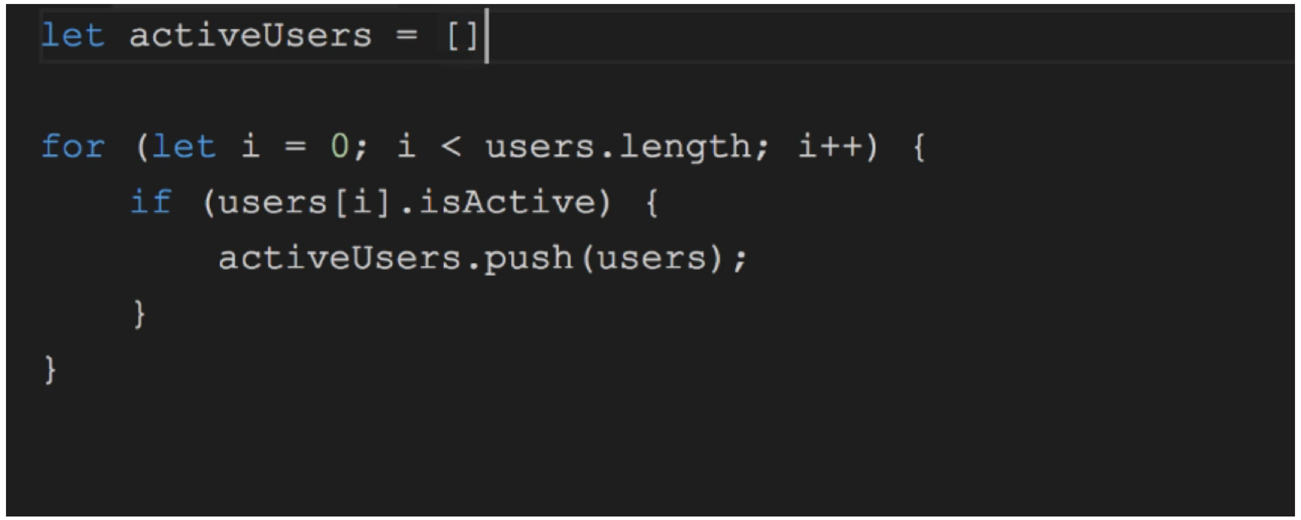
Power of Functions

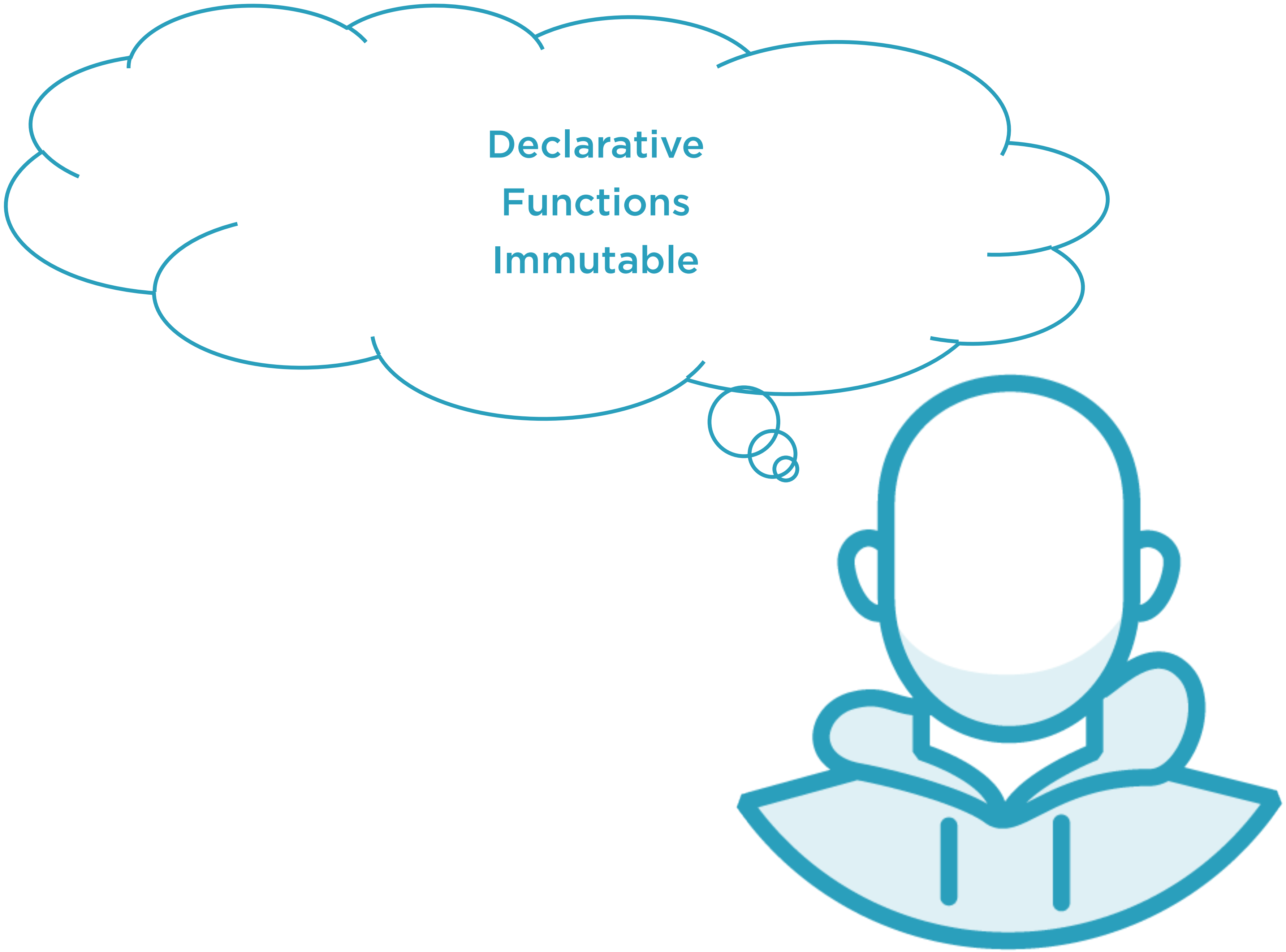
Nathan Taylor
SOFTWARE ENGINEER

@taylonr taylonr.com



```
let activeUsers = []

for (let i = 0; i < users.length; i++) {
    if (users[i].isActive) {
        activeUsers.push(users);
    }
}
```


Declarative
Functions
Immutable

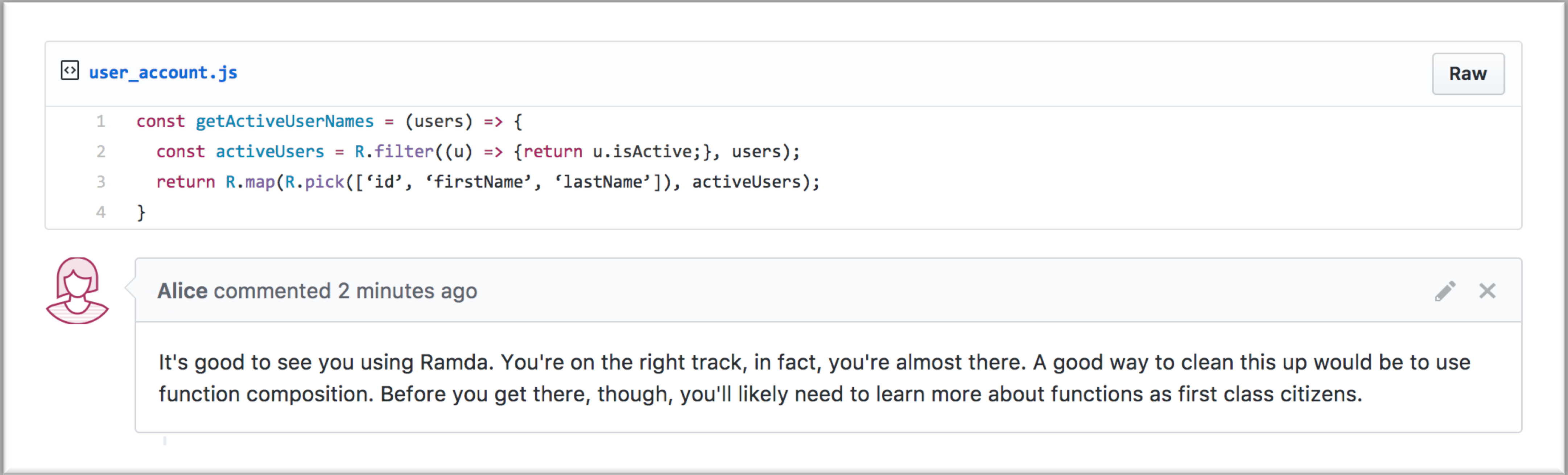
Clean declarative code is easy to reason about

IDs for Active Users

```
const getActiveUserNames = (users) => {
   const activeUsers = R.filter((u) => {
      return u.isActive;
   }, users);

   return R.map(R.pick([‘id’, ‘firstName’, ‘lastName’]),
   activeUsers);
}
```

Active User PR



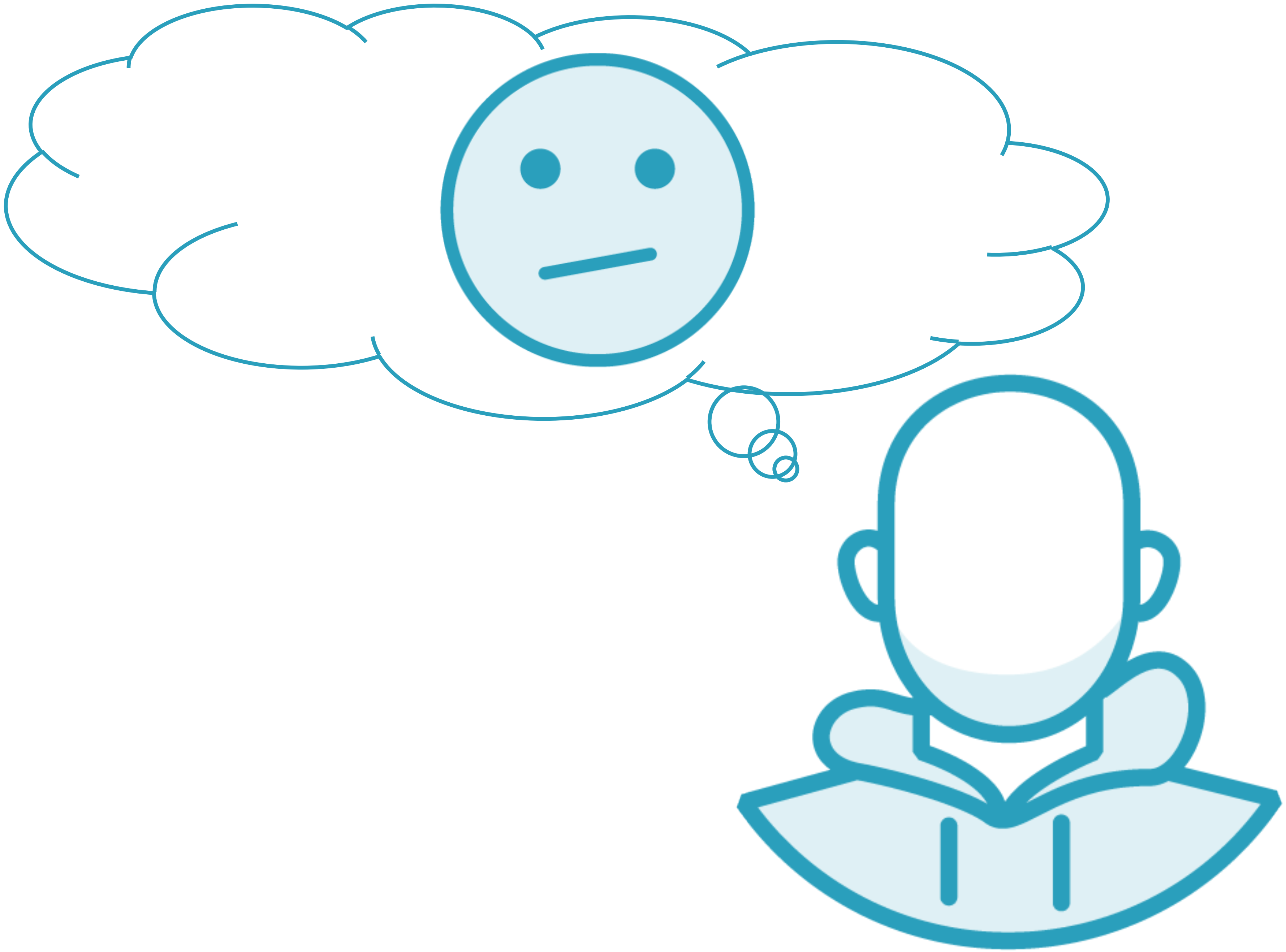
user_account.js

Raw

```
const getActiveUserNames = (users) => {
  const activeUsers = R.filter((u) => {return u.isActive;}, users);
  return R.map(R.pick([‘id’, ‘firstName’, ‘lastName’]), activeUsers);
}
```

Alice commented 2 minutes ago

It's good to see you using Ramda. You're on the right track, in fact, you're almost there. A good way to clean this up would be to use function composition. Before you get there, though, you'll likely need to learn more about functions as first class citizens.



First Class Citizens

$.ajax

```
$.ajax('users').done(function(data){});
```

Callbacks demonstrate first class functions

```
promise.then((data) => { });
```

```
it('testcase', () => {})
```

- Then takes a function
- Tests take a function as the second parameter

Declarative is Easier to Reason About

```
arr.filter(function)
```

Filter signature

```
products.filter((p) => {return p.isActive});
```

Get active products

```
const isActive = (item) => { return item.active; }
products.filter(isActive);
products.filter(isActive);

users.filter(isActive);

metadata.filter(isActive);

locations.filter(isActive);
```

◀ Reusing isActive

You have already been treating functions as first class citizens.

Returning Functions

```
users.find((u) => {
    return u.id === id;
});
```

Find takes a function

```
const byId = (item) => {
return item.id === id;}

users.find(byId);
```

◀ id is undefined

```
const byId = (item, id) => {
  return item.id === id;
}
```

```
const byId = (item, id) => {
  return item.id === id;
}

users.find((item) => {
    return byId(item, 2);
}
```

- Takes 2 parameters
- *Might* increase readability

```
const byId = (id) => {
    return (item) => {
        return item.id === id;
    }
}
```

Returning a function

```
users.find(byId(2));
```

```
const byId = (id) => {
    return (item) => {
        return item.id === 2;
    }
}
users.find(byId(2));
users.find(byId(2));
```

```
users.find((item) => {
    return item.id === 2;
});
```

Currying

Convert a function that accepts multiple parameters into a series of functions that each only take 1 parameter

```
byId(2)(item);
```

```
const deactivateUser = (id) => {
    const currentUser = byId(id);
    users.find(currentUser).active = false;
}
```

```
const deactivateUser = (id) => {
    const currentUser = byId(id);
    users.find(currentUser).active = false;
}
```

Curry

Converting a function into a series of functions

Partial Application

Supplying less arguments than required

Being able to distinguish currying and partial application is less important than being able to use functions that don’t yet have all the parameters.

```
const byId = R.curry((id, item) => {
 return item.id === id;
});


users.find(byId(2));
```

Ramda’s curry function

Curry a Function with Multiple Arguments

```
const add = R.curry((a, b, c) => {
   return a + b + c;
});

add(1)(2)(3);
```

Create helper functions to make your code more readable.

Pure Functions

What vs. How

```
users.find(byId(2));
```

Using functions in a declarative style

Pure Function

1. Doesn’t depend on any data other than what it’s passed
2. Doesn’t modify any data other than what they return

```
add(1, 2);
```

Add is a pure function

```
R.pluck(‘id’, [{id: 1}, {id: 2}]);
```

Pluck is a pure function

Is it pure? How do you use the return value?

```
R.pluck('id', [{
        id: 1
    }, {
        id: 2
    }]);
```

◀ Always returns [1, 2]

```
let users = [{
    id: 1
}, {
    id: 2
}];

users = users.push({id: 3});
```

◀ `users` equals 3

```
let users = [{
    id: 1
}, {
    id: 2
}];
users.push({id: 3});
users.push({id: 3});
```

◀ Returns 3

◀ Returns 4

Is it pure? Does it have a return value?

```
gym.addOccurrence({date: new
Date(2017, 03, 01)});
```

◂ Modifies an internal array

You supply the data

```
occurrences = list.add({date: new Date(2017, 03, 01)},
occurrences);
```

More clear what the function is doing

You have the power to write pure functions

Function Composition

```
const diff = difference(1, 2);

const val = abs(diff)

abs(difference(1,2))
```

◀ -1

◀ 1

◀ 1

Composition: combining functions

```
const absoluteDifference = (first, second) => {
    return abs(difference(first, second));
}
```

Naming a composed function

Any of the Fields are...

Null

Empty strings

Empty spaces

```
const isEmptyString = (str) => {
   const value = R.defaultTo('', str);
   const trimmedValue = R.trim(value);
   return R.isEmpty(trimmedValue);
}
```

```
const isEmptyString = (str) => {
   const value = R.defaultTo('', str);
   const trimmedValue = R.trim(value);
   return R.isEmpty(trimmedValue);
}

R.any(isEmptyString, [oldPassword, newPassword,
confirmedPassword]);
```

```
const isEmptyString = (str) => {
    return R.isEmpty(R.trim(R.defaultTo(‘’, str)));
}
```

Pipe

```
const isEmptyString = R.pipe(
    R.defaultTo(''),
    R.trim,
    R.isEmpty
);

isEmptyString('abc');
```

Pipe

```
const isEmptyString = R.pipe(
    R.defaultTo(''),
    R.trim,
    R.isEmpty
);

isEmptyString('abc');
```

```
isEmptyString = R.pipe(
 R.defaultTo(''),
 R.trim,
 R.isEmpty
);
```

```
isEmptyString = R.compose(
    R.isEmpty,
    R.trim,
    R.defaultTo('')
);
```

Summary

Step 1: Assign a function to a variable

```
const isActive = (item) => {return item.isActive;}

const selectUserNames = R.pick([‘id’, ‘firstName’,
‘lastName’]);


const getActiveUserNames = (users) => {

    const activeUsers = R.filter(isActive, users);

    return R.map(selectUserNames, activeUsers);

}
```

Step 2: Function composition

```
const isActive = (item) => {return item.isActive;}

const selectUserNames = R.pick([‘id’, ‘firstName’,
‘lastName’]);


const getActiveUserNames = R.pipe(

 R.filter(isActive),

 R.map(selectUserNames)

)(users);
```

```
const isActive = (item) => {return item.isActive;}

const selectUserNames = R.pick([‘id’, ‘firstName’,
‘lastName’]);


const getActiveUserNames = R.pipe(

 R.filter(isActive),

 R.map(selectUserNames)

)(users);
```

```
const isActive = (item) => {return item.isActive;}

const selectUserNames = R.pick([‘id’, ‘firstName’,
‘lastName’]);


const getActiveUserNames = R.pipe(

 R.filter(isActive),

 R.map(selectUserNames)

)(users);
```

```
const isActive = (item) => {return item.isActive;}

const selectUserNames = R.pick([‘id’, ‘firstName’,
‘lastName’]);


const getActiveUserNames = R.pipe(
 R.filter(isActive),
 R.map(selectUserNames)
)(users);
```

```
const isActive = (item) => {return item.isActive;}

const selectUserNames = R.pick([‘id’, ‘firstName’,
‘lastName’]);


const getActiveUserNames = R.pipe(

 R.filter(isActive),

 R.map(selectUserNames)

)(users);
```