⚙️ Functions

functions everywhere!

```
function sum (a, b) {
  return a + b
}
```

```
function sum (a, b) {
  return a + b
}

const sum = (a, b) => {
  return a + b
}
```

block body

`return` required

```
function sum (a, b) {
  return a + b
}
```

```
const sum = (a, b) => {
  return a + b
}
```

block body

`return` required

```
const sum = (a, b) => a + b
```

implicit return!

```
const sum = (a, b) => a + b
const sum = (a, b) => a + b
```

parameters

```
sum(5, 6)
```

arguments

😡 Parameters and Arguments

…for functions 😅

Arity

Arity: refers to the number of arguments that a function takes

```
const sum (a, b) => a + b
sum.length // arity of 2
```

```
const createAddress = (num, street, type) => {
  return `${num} ${street} ${type}`
}
createAddress.length // arity of 3
```

Variadic

Variadic: a “variadic” function can receive ANY number of arguments

```
function sumAll () {
  return Array.prototype.slice.call(arguments)
    .reduce((sum, next) => sum + next, 0)
}
```

```
function sumAll () {
  return Array.prototype.slice.call(arguments)
    .reduce((sum, next) => sum + next, 0)
}
```

```
function sumAll() {
  return Array.prototype.slice.call(arguments)
    .reduce((sum, next) => sum + next, 0)
}
```

🤮

```
const sumAll = (...args) => {
  return args
    .reduce((sum, next) => sum + next, 0)
}
```

```
function sumAll() {
  return Array.prototype.slice.call(arguments)
    .reduce((sum, next) => sum + next, 0)
}
```

🤮

```
const sumAll = (...args) => {
  return args
    .reduce((sum, next) => sum + next, 0)
}
```

😍

Unary

Unary: a "unary" function only takes only argument.

A "binary" function takes two arguments. A "ternary" function takes 3 arguments. And after 3…we usually say *n-ary (ex. 4-ary, 5-ary)*

🤔 Higher Order Functions

don’t be afraid of heights

Higher order function: a function that takes a function as an argument and/or returns a function

```
const some = (arr, callback) => {
  for (let i = 0; i < arr.length; i++) {
    if (callback(arr[i])) return true
  }
  return false
}
```

a higher order func!

```
const createAdder = (x) => {
  return (y) => {
    return x + y
  }
}
```

also a higher order func!

```
const some = (arr, callback) => {
  for (let i = 0; i < arr.length; i++) {
    if (callback(arr[i])) return true
  }
  return false
}

some([1, 2, 3], (n) => n < 2) // true

some([1, 2, 3], (n) => n > 4) // false
```

```
const createAdder = (x) => {
  return (y) => {
    return x + y
  }
}

const add3 = createAdder(3)
add3(5) // 8

const add6 = createAdder(6)
add6(5) // 11
```

⌨️ Documenting Types

like a functional programmer

```
const addOne = (num) => num + 1
```

`has type of`   `returns`

```
// addOne :: Int -> Int
const addOne = (num) => num + 1
const yell = (num) => num + ‘!’
```

```
// yell :: Int -> String
const yell = (num) => num + ‘!’
const hasName = (obj) => !!obj.name
```

```
// hasName :: Object -> Bool
const hasName = (obj) => !!obj.name
const names = (arrOfPersons) =>
  arr.map(person => person.name)
```

```
// names :: [Object] -> [String]
const names = (arrOfPersons) =>
  arr.map(person => person.name)
```

```
const getProperty = (obj) => {
  return (key) => {
    return obj[key]
  }
}
```

```
// names :: Object -> String -> *
const getProperty = (obj) => {
  return (key) => {
    return obj[key]
  }
}
```

```
// names :: Object -> String -> *
const getProperty = (obj) => (key) => obj[key]
```

🎶 🎵 Composition

building functions from functions

🎶 🎵 Composition

- Glue small functions together to make big functions
- Two ways: A) manually, B) using a helper function

```
const inc  = x => x + 1
const yell = s => s + '!'
```

🎶 🎵 Composition

- Glue small functions together to make big functions
- Two ways: A) manually, B) using a helper function

```
const inc  = x => x + 1
const yell = s => s + '!'

// manually
const yellIncA = x => yell(inc(x))
```

🎶 🎵 Composition

- Glue small functions together to make big functions
- Two ways: A) manually, B) using a helper function

```
const inc  = x => x + 1
const yell = s => s + '!'

// manually
const yellIncA = x => yell(inc(x))

// with a helper
const yellIncB = compose(yell, inc)
```

🎶 🎵 Composition

- Glue small functions together to make big functions
- Two ways: A) manually, B) using a helper function

```
const inc  = x => x + 1
const yell = s => s + '!'
```

func that does `inc` then `yell`

```
// manually
const yellIncA = x => yell(inc(x))
```

func that does `inc` then `yell`

```
// with a helper
const yellIncB = compose(yell, inc)

yellIncA(7) // '8!'
yellIncB(7) // '8!'
```

🎶🎵 Composition

```
                                7
(x => yell(inc(x))) (7)

compose(yell, inc)  (7)
                     7
```

🎶🎵 Composition

```
                    8
(x => yell(inc(x))) (7)

compose(yell, inc)  (7)
               8
```

🎶 🎵 Composition

```
        '8!'
(x => yell(inc(x))) (7)


compose(yell, inc)  (7)
          '8!'
```

⬅️ Composition flows *right to left*

```
(x => yell(inc(x))) (7)

compose(yell, inc)  (7)
```

$$yell \circ inc$$

$$\xleftarrow{\qquad\qquad}$$

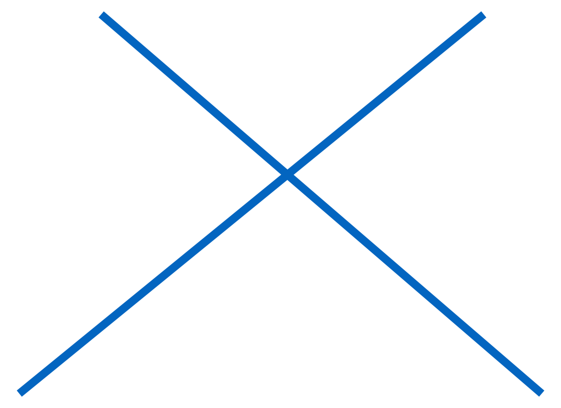
➡️ Pipe flows *left to right*

```
                            7
(x => yell(inc(x))) (7)


pipe(inc, yell)     (7)
                     7
```

➡️ Pipe flows *left to right*

```
                  8
(x => yell(inc(x))) (7)
```



```
pipe(inc, yell)     (7)
        8
```

➡️ Pipe flows *left to right*

```
      '8!'
(x => yell(inc(x))) (7)

pipe(inc, yell)     (7)
           '8!'
```

➡️ Pipe flows *left to right*

```
(x => yell(inc(x))) (7)

pipe(inc, yell)     (7)
```

$$\overrightarrow{inc \ggg yell}$$

Composition/Piping is Associative…

🟦 🟧 🟪

```
compose(compose(yell, inc), double)(7) // '15!'

compose(yell, compose(inc, double))(7) // '15!'
```

🟦 🟧 🟪

$$(yell \circ inc) \circ double$$

$$= yell \circ (inc \circ double)$$

definition of associativity

…but a good JS `compose` is also variadic.

"can take a variable number of arguments"

```
compose(yell, inc, double)(7) // '15!'
```

⬅️ Variadic Compose vs. ➡️ Variadic Pipe

a function!

```
const composedA = arr => yell(double(length(arr)))
```

←

the same function!

```
const composedB = compose(yell, double, length)
```

←

also the same!

```
const pipeline = pipe(length, double, yell)
```

→

Why?

Build Complex Fn from Simpler Fns

```
const listInstructorsWhoCanJoinRoadtrip = pipe(
    sortDescendingByDate,
    removeDuplicatesById,
    takeFreeInstructors
)

listInstructorsWhoCanJoinRoadtrip(rawData)
// 'Karen', 'Ben', 'Gabriel'
```

Testing

- Small, pure functions are easy to test
- If you can trust the small functions, you can trust their composition

Think in Terms of Types

String → Number >>> Number → *[Number]* >>> *[Number]* → Object

=

String → *[Number]* >>> *[Number]* → Object

=

String → Object

OK, where can I get `compose`/`pipe`?

Write it yourself!

…or `const { pipe } = require('ramda')`

Currying 💗

and partial application

Calling Func with Same Args

```
// gFetch takes domain, method, route, body
const result1 = await gFetch('http://coolsite.com', 'GET', '/', null)
```

Calling Func with Same Args

```
// gFetch takes domain, method, route, body
const result1 = await gFetch('http://coolsite.com', 'GET', '/', null)
const result2 = await gFetch('http://coolsite.com', 'GET', '/puppies', null)
```

Calling Func with Same Args

what a drag!

```
// gFetch takes domain, method, route, body
const result1 = await gFetch('http://coolsite.com', 'GET', '/', null)
const result2 = await gFetch('http://coolsite.com', 'GET', '/puppies', null)
const result3 = await gFetch('http://coolsite.com', 'POST', '/puppies', puppy)
```

Calling Func with Same Args

```
// gFetch takes domain, method, route, body
const result1 = await gFetch('http://coolsite.com', 'GET', '/', null)
const result2 = await gFetch('http://coolsite.com', 'GET', '/puppies', null)
const result3 = await gFetch('http://coolsite.com', 'POST', '/puppies', puppy)


// wouldn't this be nice?
const gFetchCool = gFetch('http://coolsite.com')
```

Calling Func with Same Args

```
// gFetch takes domain, method, route, body
const result1 = await gFetch('http://coolsite.com', 'GET', '/', null)
const result2 = await gFetch('http://coolsite.com', 'GET', '/puppies', null)
const result3 = await gFetch('http://coolsite.com', 'POST', '/puppies', puppy)
```

from gFetch we made gFetchCool...

whoah, only one argument?

```
// wouldn't this be nice?
const gFetchCool = gFetch('http://coolsite.com')
```

Calling Func with Same Args

```
// gFetch takes domain, method, route, body
const result1 = await gFetch('http://coolsite.com', 'GET', '/', null)
const result2 = await gFetch('http://coolsite.com', 'GET', '/puppies', null)
const result3 = await gFetch('http://coolsite.com', 'POST', '/puppies', puppy)
```

from gFetch we made gFetchCool...

whoah, only one argument?

```
// wouldn't this be nice?
const gFetchCool = gFetch('http://coolsite.com')
const result4 = await gFetchCool('GET', '/', null)
const result5 = await gFetchCool('GET', '/puppies', null)
```

...so we have to provide three more

Calling Func with Same Args

```
// gFetch takes domain, method, route, body
const result1 = await gFetch('http://coolsite.com', 'GET', '/', null)
const result2 = await gFetch('http://coolsite.com', 'GET', '/puppies', null)
const result3 = await gFetch('http://coolsite.com', 'POST', '/puppies', puppy)


// wouldn't this be nice?
const gFetchCool = gFetch('http://coolsite.com')
const result4 = await gFetchCool('GET', '/', null)
const result5 = await gFetchCool('GET', '/puppies', null)



const createPuppy = gFetchCool('POST', '/puppies')
```

Calling Func with Same Args

```
// gFetch takes domain, method, route, body
const result1 = await gFetch('http://coolsite.com', 'GET', '/', null)
const result2 = await gFetch('http://coolsite.com', 'GET', '/puppies', null)
const result3 = await gFetch('http://coolsite.com', 'POST', '/puppies', puppy)


// wouldn't this be nice?
const gFetchCool = gFetch('http://coolsite.com')
const result4 = await gFetchCool('GET', '/', null)
const result5 = await gFetchCool('GET', '/puppies', null)
```

…& from that we made createPuppy two more arguments (= three total)

```
const createPuppy = gFetchCool('POST', '/puppies')
```

Calling Func with Same Args

```
// gFetch takes domain, method, route, body
const result1 = await gFetch('http://coolsite.com', 'GET', '/', null)
const result2 = await gFetch('http://coolsite.com', 'GET', '/puppies', null)
const result3 = await gFetch('http://coolsite.com', 'POST', '/puppies', puppy)


// wouldn't this be nice?
const gFetchCool = gFetch('http://coolsite.com')
const result4 = await gFetchCool('GET', '/', null)
const result5 = await gFetchCool('GET', '/puppies', null)
```

...& from that we made createPuppy two more arguments (= three total)

```
const createPuppy = gFetchCool('POST', '/puppies')
const result6 = await createPuppy(puppy1)
const result7 = await createPuppy(puppy2)
```

...leaving just one argument to go

💗 Currying Facts

- **A staple of FP**
  - In purely functional languages, all functions are usually curried by default
  - In JS, functions are **not** curried by default
    - “Jack of all trades, master of none”
    - However, libraries can help us out here!

- **Lets you *partially apply* functions (only pass in *some* args)**
  - You get back a function "waiting" for more arguments
  - When you finally supply *all* arguments, you get a result

The Simple Way

```
// before
const greet = (salutation, excited, name) =>
    `${salutation}, ${name}${excited ? '!' : '.'}`

greet('Hello', true, 'Kate') // 'Hello, Kate!'
greet('Hello', true, 'Dan')  // 'Hello, Dan!'
```

function that returns a function that returns a function

```
// after
const greet = salutation => excited => name =>
    `${salutation}, ${name}${excited ? '!' : '.'}`

greet('Hello')(true)('Kate') // 'Hello, Kate!'
greet('Hello')(true)('Dan')  // 'Hello, Dan!'
```

have to invoke each returned function separately

Why?

Derive Specific Fns from General Fn

```
const exclaimHi = greet('Hi')(true)
exclaimHi('Karen') // 'Hi, Karen!'
exclaimHi('Matt')  // 'Hi, Matt!'

const sayBye = greet('Bye')(false)
sayBye('Gabriel')          // 'Bye, Gabriel.'
sayBye('imperative code') // 'Bye, imperative code.'
```

good for reuse!

good for higher-order functions!

Currying: ❤️ *n*-ary func → 💗 *n* unary funcs

started with a ternary (3-ary) function

```
// greet ::   (String,      Bool,  String) -> String
const greet = (salutation, excited, name) => `...`
```

converted it to 3 nested functions

```
// greet ::   String     ->  Bool     ->  String -> String
const greet = salutation =>  excited =>  name => `...`
```

(function definition is right-associative)

But… this looks weird!

pretty verbose, lots of parens

```
// current solution
const result = greet('Hi')(true)('Ashi')
```

```
// desired solution can still do this:
const result = greet('Hi', true, 'Ashi')
```

ah, that's idiomatic JS

Solution: *FANCY* Currying

```
const curriedGreet = curry(greet)
```

Solution: *FANCY* Currying

```
const curriedGreet = curry(greet)
curriedGreet('Hi', true, 'Ashi')
```

Solution: *FANCY* Currying

```
const curriedGreet = curry(greet)
curriedGreet('Hi', true, 'Ashi')
curriedGreet('Hi', true)('Ashi')
```

Solution: *FANCY* Currying

```
const curriedGreet = curry(greet)
curriedGreet('Hi', true, 'Ashi')
curriedGreet('Hi', true)('Ashi')
curriedGreet('Hi')(true, 'Ashi')
```

Solution: *FANCY* Currying

higher-order helper function

```
const curriedGreet = curry(greet)
curriedGreet('Hi', true, 'Ashi')
curriedGreet('Hi', true)('Ashi')
curriedGreet('Hi')(true, 'Ashi')
curriedGreet('Hi')(true)('Ashi')
```

resulting function can accept
any number of args at a time!

OK, where can I get `curry`?

Write it yourself!

…or `const { curry } = require('ramda')`

💟 Putting It All Together

Higher-Order Fns, Composition, & Partial Application

Opportunity for Composition

```
const yellUpper = string => yell(upper(string))

                  param  =>    f(    g(param ))

                  compose(     f,    g)

const yellUpper = compose(yell, upper)
```

Opportunity for Partial Application

assuming `map` is curried...

```
const capitalizeAll = strings => map(upper, strings)

                      param   =>   f(arg1,  param   )

                                   f(arg1)

const capitalizeAll =             map(upper)
```

From Point-ful to Point-free

("point"s are parameters)

```
const getShortNames = people => map(person => shorten(toName(person)), people)
```

use composition

~~param~~ => f(g(~~param~~))

```
compose( f, g)
```

```
const getShortNames = people => map(compose(shorten, toName), people)
```

use partial application

~~param~~ => f(arg1, ~~param~~)

```
f(arg1)
```

```
const getShortNames = map(compose(shorten, toName))
```

Why?

Less Noise

```
const getShortNames = people => map(person => shorten(toName(person)), people)
```

vs

```
const getShortNames = map(pipe(toName, shorten))
```

Amar Shah's Rules*

1. **Use** point-free style **when** it communicates better.
2. **Avoid** point-free style **when** it doesn't.

*YouTube: Point Free or Die

WORKSHOP