What the Functional?

(What, Why, and How)

What the Functional?

(What, Why, and How)

Rob Howard
@damncabbage
http://robhoward.id.au

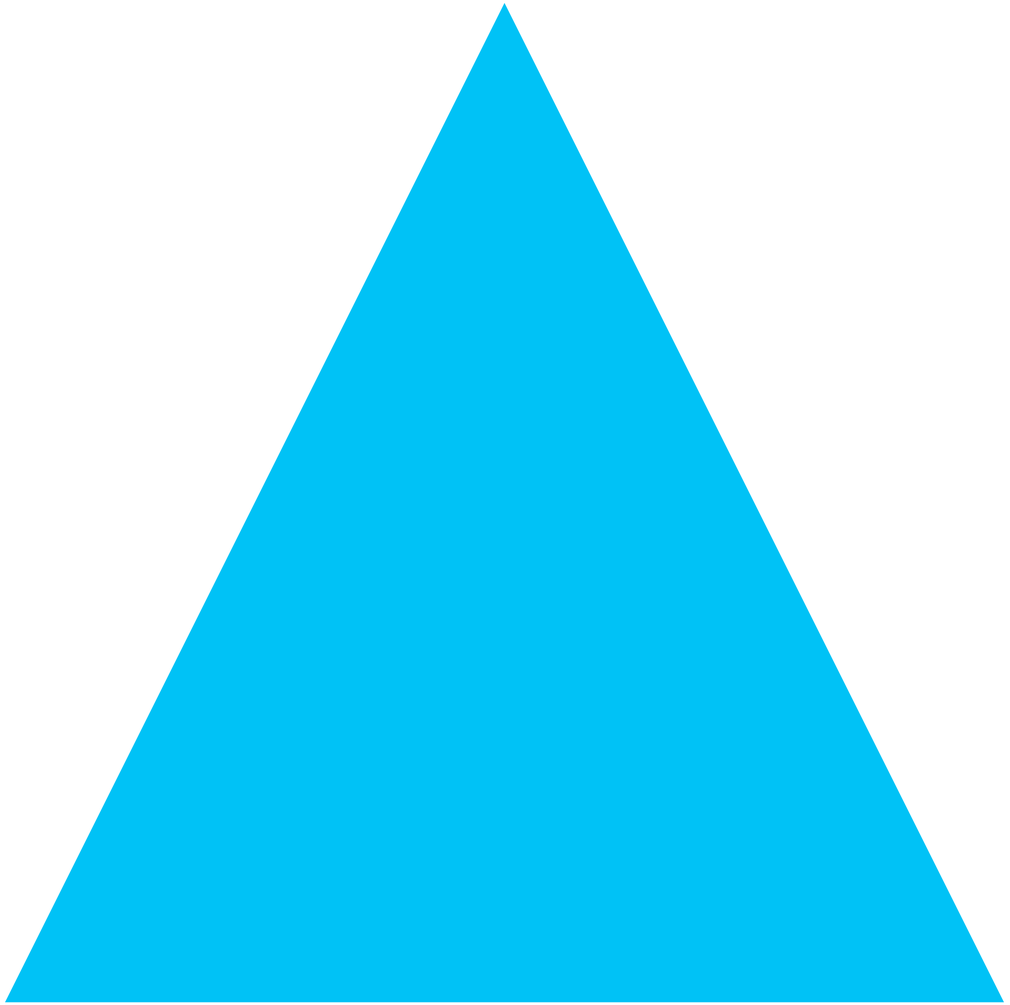
What the Functional?

(What, Why, and How)

Rob Howard
@damncabbage
http://robhoward.id.au

Let's Get Meta

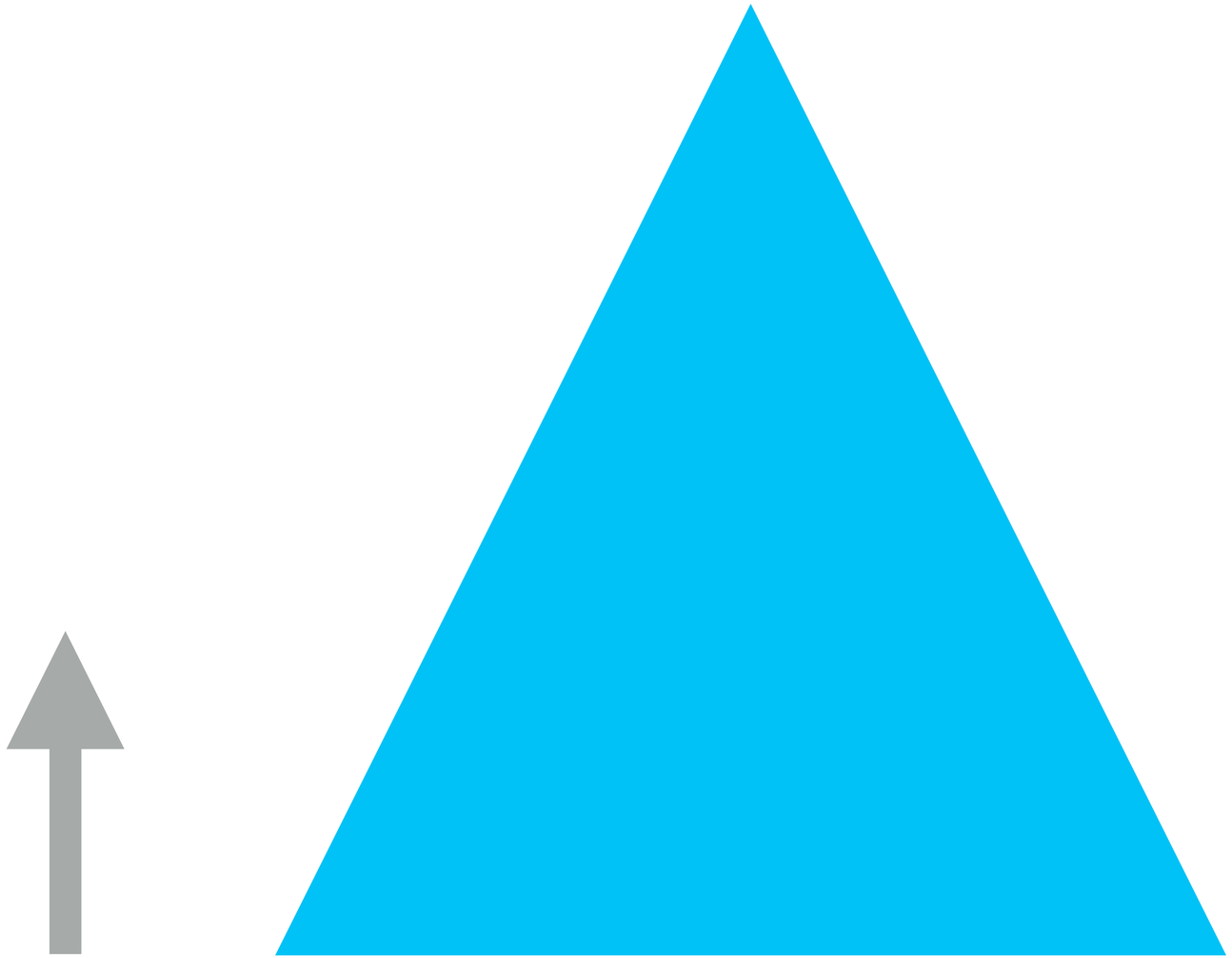
Jargon



The

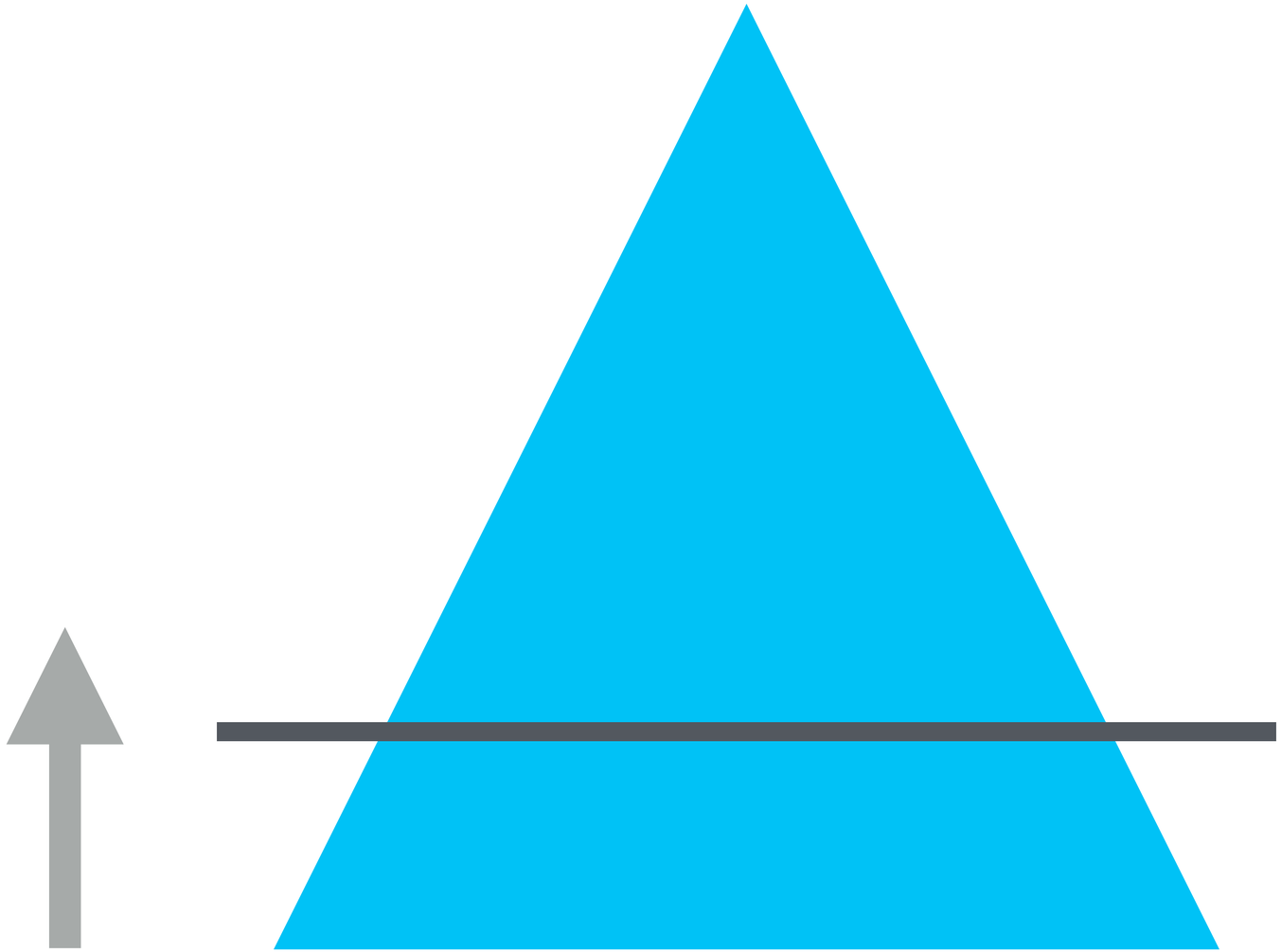
Functional Programming Pyramid

(Courtesy of / stolen from Lucas Di Cioccio.)



The
Functional Programming Pyramid

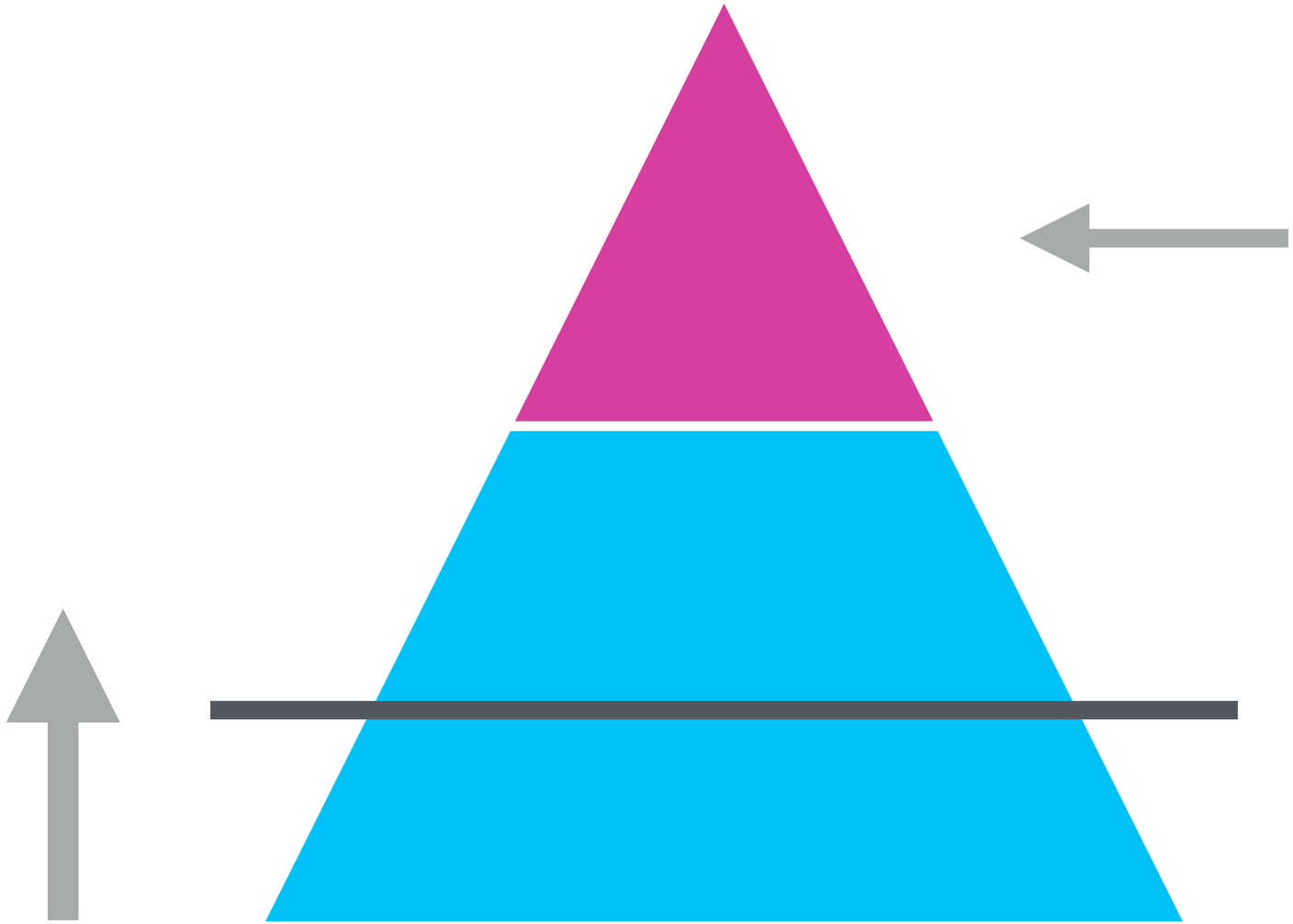
(Courtesy of / stolen from Lucas Di Cioccio.)



The
Functional Programming Pyramid

"Productive FP" bar

(Courtesy of / stolen from Lucas Di Cioccio.)



What gets discussed on social networks

"Productive FP" bar

The
Functional Programming Pyramid

(Courtesy of / stolen from Lucas Di Cioccio.)


Disneyland
MICKEY'S TOONTOWN
BIG THUNDER RANCH
CRITTER COUNTRY
FANTASYLAND
PLAZA GARDENS STAGE
TOMORROWLAND TERRACE
FRONTIERLAND
RIVERS OF AMERICA
TOMORROWLAND
NEW ORLEANS SQUARE
PLAZA PAVILLION
ADVENTURELAND
MAIN STREET, U.S.A.
CITY HALL
MICKEY & FRIENDS TRAM ROUTE
DOWNTOWN DISNEY® DISTRICT AND HOTELS OF THE DISNEYLAND® RESORT
DISNEY'S CALIFORNIA ADVENTURE
LION KING TRAM ROUTE

So. Why?

```
function unblockAll(batchSize, authDetails) {
  return makeTwitterClient(authDetails)
    .chain(client =>
      getBlocksList(client)
        .map(response => response.ids.map(
          id => unblockOne(client, id)
        )
    )
    .chain(reqs =>
      Future.parallel(batchSize, reqs)
    )
}
```

What is
"Functional Programming"?

the universal property

$$\forall h \in \langle\!\langle \mu F \to \sigma \rangle\!\rangle \,.\; h = \mathit{fold}_F\ f \;\Leftrightarrow\; h \circ \mathit{in}_{\mu F} = f \circ F\ h. \quad (18)$$

This is how $\langle\!\langle \mathsf{fold}_F \rangle\!\rangle$ is defined. To define $\langle\!\langle \mathsf{unfold}_F \rangle\!\rangle$ we go via the *final* object $\mathit{out}_{\nu F}$ in F-$\mathsf{Coalg}(\mathsf{SET})$ (the category of F-coalgebras) instead. The semantics of all terms are given in Figure 6.

The domain-theoretic semantics lives in the category CPO of CPOs and continuous functions. To define $[\![\mu F]\!]$, the category $\mathsf{CPO}_\perp$ of CPOs and *strict* continuous functions is also used. We want all types in the domain-theoretic semantics to be lifted (like in Haskell). To model this we lift all functors using L, which is defined in Figure 5.

If we were to define $[\![\mathsf{fold}_F]\!]$ using the same method as for $\langle\!\langle \mathsf{fold}_F \rangle\!\rangle$, then that would restrict its arguments to be strict functions. An explicit fixpoint is used instead. The construction still satisfies the universal property associated with folds if all functions involved are strict [FM91]. For symmetry we also define $[\![\mathsf{unfold}_F]\!]$ using an explicit fixpoint; that does not affect its universality property.

The semantics of fix is, as usual, given by a least fixpoint construction.

We have been a little sloppy above, in that we have not defined the action of the functor K_σ on objects. When working in SET we let $K_\sigma\ A = \langle\!\langle \sigma \rangle\!\rangle$, and in CPO and $\mathsf{CPO}_\perp$ we let $K_\sigma\ A = [\![\sigma]\!]$. Otherwise the functors have their usual meanings.

4. Partial equivalence relations

In what follows we will use *partial equivalence relations*, or PERs for short.

A PER on a set S is a symmetric and transitive binary relation on S. For a PER R on S, and some $x \in S$ with xRx, define the *equivalence class* of x as

$$[x]_R = \{\, y \,|\, y \in S,\ xRy \,\}. \quad (19)$$

(The index R is omitted below.) Note that the equivalence classes partition $\mathrm{dom}(R) = \{\, x \in S \,|\, xRx \,\}$, the *domain* of R. Let $[R]$ denote the set of equivalence classes of R.

For convenience we will use the notation $\{c\}$ for an arbitrary element $x \in c$, where c is an equivalence class of some PER $R \subseteq S^2$. This definition is of course ambiguous, but the ambiguity disappears in many contexts, including in Section 5, we have that $[\mathit{ini}(\{c\})]$ denotes the same equivalence class no matter which element in c is chosen.

5. Moral equality

We will now inductively define a family of PERs $\sim_\sigma$ on the domain-theoretic semantic domains; with $\mathit{Rel}(\sigma) = \wp([\![\sigma]\!]^2)$ we will have $\sim_\sigma \in \mathit{Rel}(\sigma)$. (Here $\wp(X)$ is the power set of X. The index σ will sometimes be omitted.)

If two values are related by $\sim$, then we say that they are *morally equal*. We use moral equality to formalise totality: a value $x \in [\![\sigma]\!]$ is said to be *total* iff $x \in \mathrm{dom}(\sim_\sigma)$. The intention is that if σ does not contain function spaces, then we should have $x \sim_\sigma y$ iff x and y are equal, total values. For functions we will have $f \sim g$ iff f and g map (total) related values to related values.

The definition of totality given here should correspond to basic intuition. Sometimes another definition is used instead, where $f \in [\![\sigma \to \tau]\!]$ is total iff $f@x = \perp$ implies that $x = \perp$. That definition is not suitable for non-strict languages where most semantic domains are not flat. As a simple example, consider $[\![\mathsf{fst}]\!]$; we will have $[\![\mathsf{fst}]\!] \in \mathrm{dom}(\sim)$, so $[\![\mathsf{fst}]\!]$ is total according to our definition, but $[\![\mathsf{fst}]\!]@(\perp, \perp) = \perp$.

Given the family of PERs $\sim$ we can relate the set-theoretic semantic values with the total values of the domain-theoretic semantics; see Sections 6 and 7.

5.1 Non-recursive types

The PER $\sim_{\sigma\to\tau}$ is a logical relation, i.e. we have the following definition for function spaces:

$$\begin{aligned} f \sim_{\sigma\to\tau} g \;\Leftrightarrow\; & \\ & f \neq \perp \wedge\ g \neq \perp \wedge \\ & \forall x, y \in [\![\sigma]\!] \,.\; x \sim_\sigma y \;\Rightarrow\; f@x \sim_\tau g@y. \end{aligned} \quad (20)$$

We need to ensure explicitly that f and g are non-bottom because some of the PERs will turn out to have $\perp \in \mathrm{dom}(\sim)$ or $\mathrm{dom}(\sim) = \emptyset$.

Pairs are related if corresponding components are related:

$$\begin{aligned} x \sim_{\sigma\times\tau} y \;\Leftrightarrow\; & \exists x_1, y_1 \in [\![\sigma]\!]\,,\ x_2, y_2 \in [\![\tau]\!] \,. \\ & x = (x_1, x_2) \ \wedge\ y = (y_1, y_2) \ \wedge \\ & x_1 \sim_\sigma y_1 \ \wedge\ x_2 \sim_\tau y_2. \end{aligned} \quad (21)$$

… Sorry.

Let's come back to this.

Referential Transparency

1.

```
function double(x) {
  // ...
}



var r1 = double(100);
var r2 = double(100);
```

1.

```
function double(x) {
  // ...
}



var r1 = double(100);
var r2 = double(100);
```

1.

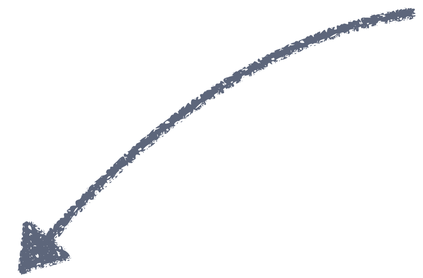
```
function double(x) {
  // ...
}


var result = double(100);
var r1 = result;
var r2 = result;
```

1.

```
function double(x) {
  // ...
}


var result = double(100);
var r1 = result;
var r2 = result;
```

1.

```
function double(x) {
  // ...
}
```



Referentially Transparent

```
var result = double(100);
var r1 = result;
var r2 = result;
```

1.

```
function double(x) {
  // ...
}



var r1 = double(100);
var r2 = double(100);
```

1.

```
function double(x) {
  // ...
}


var result = double(100);
var r1 = result;
var r2 = result;
```

1.

```
function double(x) {
  // ...
}


var result = double(100);
var r1 = result;
var r2 = result;
```

1.

```
function double(x) {
  // ...
}


var result = double(100);
var r1 = result;
var r2 = result;
```

1.

```
function double(x) {
  // ...
}


var result = double(100);
var r1 = result;
var r2 = result;
```

1.

```
function double(x) {
  // ...
}

var oneHundred = 100;
var result = double(oneHundred);
var r1 = result;
var r2 = result;
```

1.

```
function double(x) {
  // ...
}

var oneHundred = 100;
var result = double(oneHundred);
var r1 = result;
var r2 = result;
```

1.

```
function double(x) {
  // ...
}

var oneHundred = 100;
var result = double(oneHundred);
var r1 = result;
var r2 = result;
```

1.

```
function double(x) {
  return x * 2;
}

var oneHundred = 100;
var result = double(oneHundred);
var r1 = result;
var r2 = result;
```

1.

```
function double(x) {
  return x * 2;
}

var oneHundred = 100;
var result = double(oneHundred);
var r1 = result;          // 200
var r2 = result;          // 200
```

1.

```
function double(x) {
  return x * 2;
}
```

← "Pure"

```
var r1 = double(100); // 200
var r2 = double(100); // 200
```

"An expression is referentially transparent if, in a program, all occurrences of that expression can be replaced by an assignment to that expression without observably changing that program."

(Paraphrased from "Functional Programming in Scala" by Paul Chiusano and Rúnar Bjarnason)

"An expression is referentially transparent if, in a program, all occurrences of that expression can be replaced by an assignment to that expression without observably changing that program."

(Paraphrased from "Functional Programming in Scala" by Paul Chiusano and Rúnar Bjarnason)

"An expression is referentially transparent if, in a program, all occurrences of that expression can be replaced by an assignment to that expression without observably changing that program."

(Paraphrased from "Functional Programming in Scala" by Paul Chiusano and Rúnar Bjarnason)

"An expression is referentially transparent if, in a program, all occurrences of that expression can be replaced by an assignment to that expression without observably changing that program."

(Paraphrased from "Functional Programming in Scala" by Paul Chiusano and Rúnar Bjarnason)

"Pure Functions"

or

"Purity"


U.S.

And now some more examples:

2.

```
function double(x) {
  // ...
}



var r1 = double(100);
var r2 = double(100);
```

2.

```
function double(x) {
  console.log("hi")
  return x * 2;
}


var r1 = double(100);
var r2 = double(100);
```

2.

```
function double(x) {
  console.log("hi")
  return x * 2;
}


var r1 = double(100);
var r2 = double(100);
```

"Impure"

2.

```
function double(x) {
  console.log("hi")
  return x * 2;
}


var r1 = double(100);
var r2 = double(100);
```

2.

```
function double(x) {
  console.log("hi")
  return x * 2;
}
```

```
function double(x) {
  console.log("hi")
  return x * 2;
}
```

```
> double(100)
hi
=> 200
```

```
function double(x) {
  console.log("hi")
  return x * 2;
}
```

```
> double(100)
hi
=> 200
```

```
function double(x) {
  console.log("hi")
  return x * 2;
}
```

```
> double(100)
hi
=> 200
```

2.

```
function double(x) {
  console.log("hi")
  return x * 2;
}


var r1 = double(100);
var r2 = double(100);
```

2.

```
function double(x) {
  console.log("hi")
  return x * 2;
}


var r1 = double(100);
var r2 = double(100);
```

"Impure"

2.

```
function double(x) {
  console.log("hi")
  return x * 2;
}
```


```
var r1 = double(100);
var r2 = double(100);
```

2.

```
function double(x) {
  console.log("hi")
  return x * 2;
}
```


```
var r1 = double(100); // prints "hi"
var r2 = double(100); // prints "hi"
```

2.

```
function double(x) {
  console.log("hi")
  return x * 2;
}
```


```
var result = double(100); // prints "hi"
var r1 = result;
var r2 = result;
```

3.

```
function doublePlusCounter(x) {
  return x * 2 + (counter || 0);
}


var counter = 1;
var r1 = double(100); // 201
var r2 = double(100); // 201
```

3.

```
function doublePlusCounter(x) {
  return x * 2 + (counter || 0);
}


var counter = 1;
var r1 = double(100); // 201
var r2 = double(100); // 201
```

3.

```
function doublePlusCounter(x) {
  return x * 2 + (counter || 0);
}

var result = double(100);
var counter = 1;
var r1 = result; // 200
var r2 = result; // 200
```

3.

```
function doublePlusCounter(x) {
  return x * 2 + (counter || 0);
}

var result = double(100);
var counter = 1;
var r1 = result; // 200
var r2 = result; // 200
```

4.

```
function bumpCounter() {
  counter = counter + 1;
}
var counter = 1;

console.log(counter); // => 1
var r1 = bumpCounter(); // counter = 2
var r2 = bumpCounter(); // counter = 3
```

4.

```
function bumpCounter() {
  counter = counter + 1;
}
var counter = 1;

console.log(counter); // => 1
var r1 = bumpCounter(); // counter = 2
var r2 = bumpCounter(); // counter = 3
```

4.

```
function bumpCounter() {
  counter = counter + 1;
}
var counter = 1;
var result = bumpCounter();
console.log(counter); // => 2
var r1 = result; // counter = 2
var r2 = result; // counter = 2
```

4.

```
function bumpCounter() {
  counter = counter + 1;
}
var counter = 1;
var result = bumpCounter();
console.log(counter); // => 2
var r1 = result; // counter = 2
var r2 = result; // counter = 2
```

5.

```
function append(x) {
  x.push(1);
  return x;
}
var list = [1];

var r1 = append(list); // [1,1]
var r2 = append(list); // [1,1,1]
```

5.

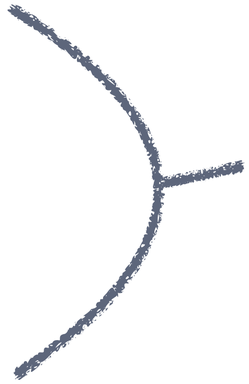
```
function append(x) {
  x.push(1);
  return x;
}
var list = [1];

var r1 = append(list); // [1,1]
var r2 = append(list); // [1,1,1]
```

5.

```
function append(x) {
  x.push(1);
  return x;
}
var list = [1];
var result = append(list);
var r1 = result; // [1,1]
var r2 = result; // [1,1]
```

5.

```
function append(x) {
  x.push(1);
  return x;
}
var list = [1];
var result = append(list);
var r1 = result; // [1,1]
var r2 = result; // [1,1]
```



"Impure"

BUT

5.

```
function append(x) {
  x.push(1);
  return x;
}
var list = [1];

var r1 = append(list); // [1,1]
var r2 = append(list); // [1,1,1]
```

5.

```
function append(x) {
  x2 = x.slice(0);
  x2.push(1); return x;
}
var list = [1];

var r1 = append(list); // [1,1]
var r2 = append(list); // [1,1]
```

5.

```
function append(x) {
  x2 = x.slice(0);
  x2.push(1); return x;
}
var list = [1];

var r1 = append(list); // [1,1]
var r2 = append(list); // [1,1]
```

Now "Pure"

So what is "Functional Programming"?

Referential Transparency

(and/or "purity")

"Functional programming is programming with pure functions."

— Rúnar Bjarnason

"Functional Languages"

What does purity get us specifically?

"The behavior of a pure function is independent of where and when it is evaluated, whereas the behavior of an impure function is intrinsically tied to its execution order."

— Stephen Diehl

1.

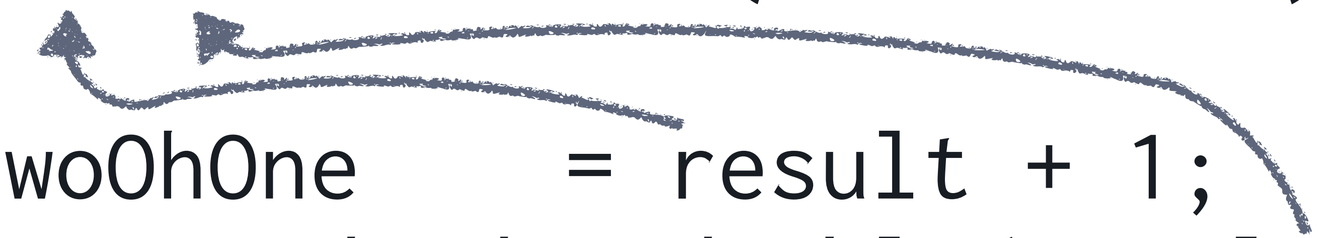
```
function double(x) {
  return x * 2;
}

var oneHundred = double(50);

var result = double(oneHundred);

var twoOhOne    = result + 1;
var fourHundred = double(result);
```

1.

```
function double(x) {
  return x * 2;
}

var oneHundred = double(50);

var result = double(oneHundred);

var twoOhOne     = result + 1;
var fourHundred = double(result);
```

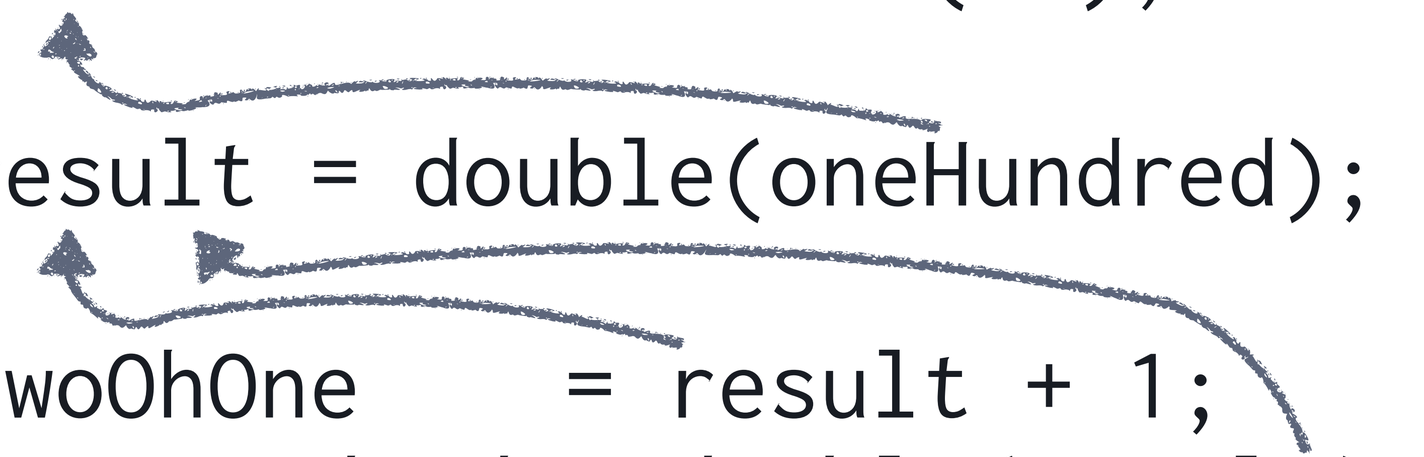
1.

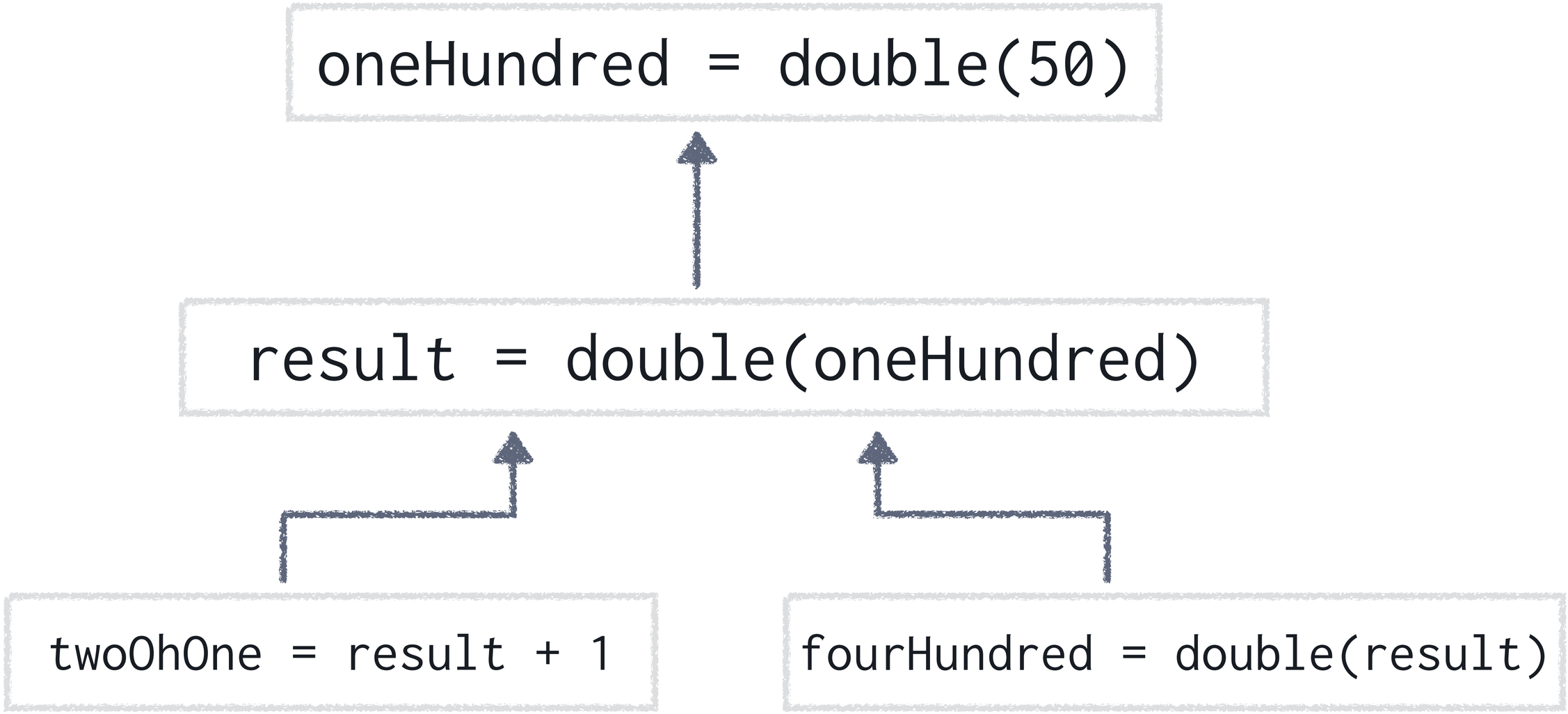
```
function double(x) {
  return x * 2;
}

var oneHundred = double(50);

var result = double(oneHundred);

var twoOhOne     = result + 1;
var fourHundred = double(result);
```

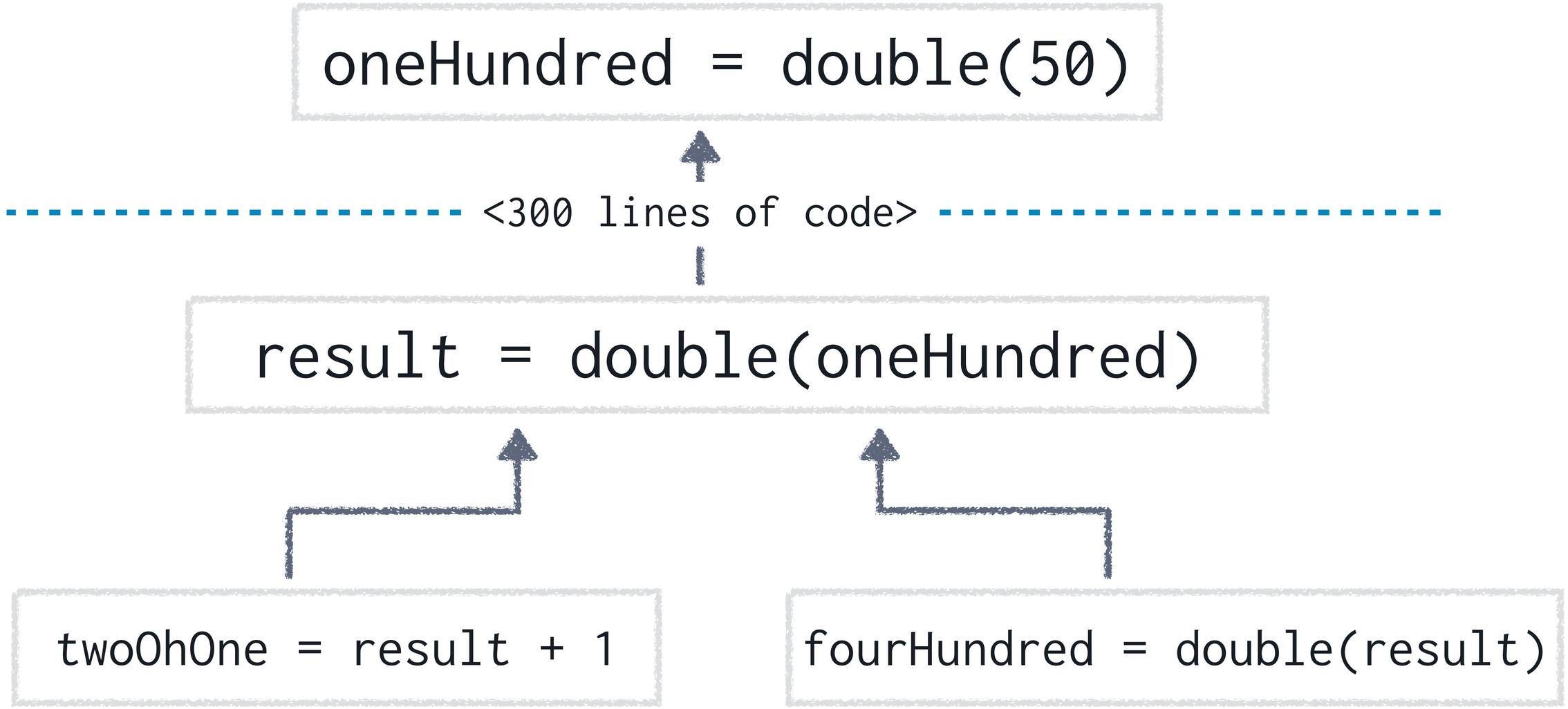


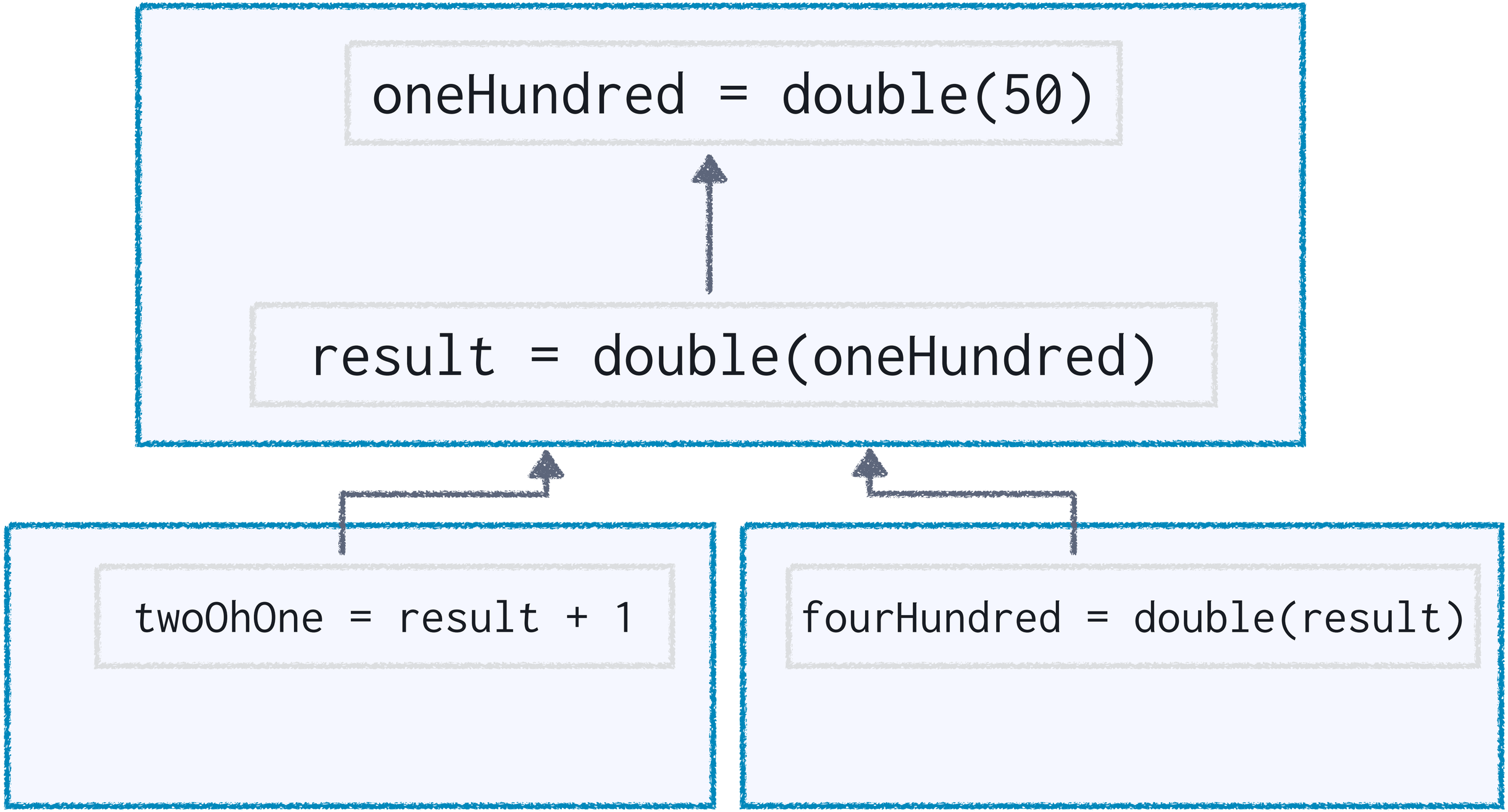
1.

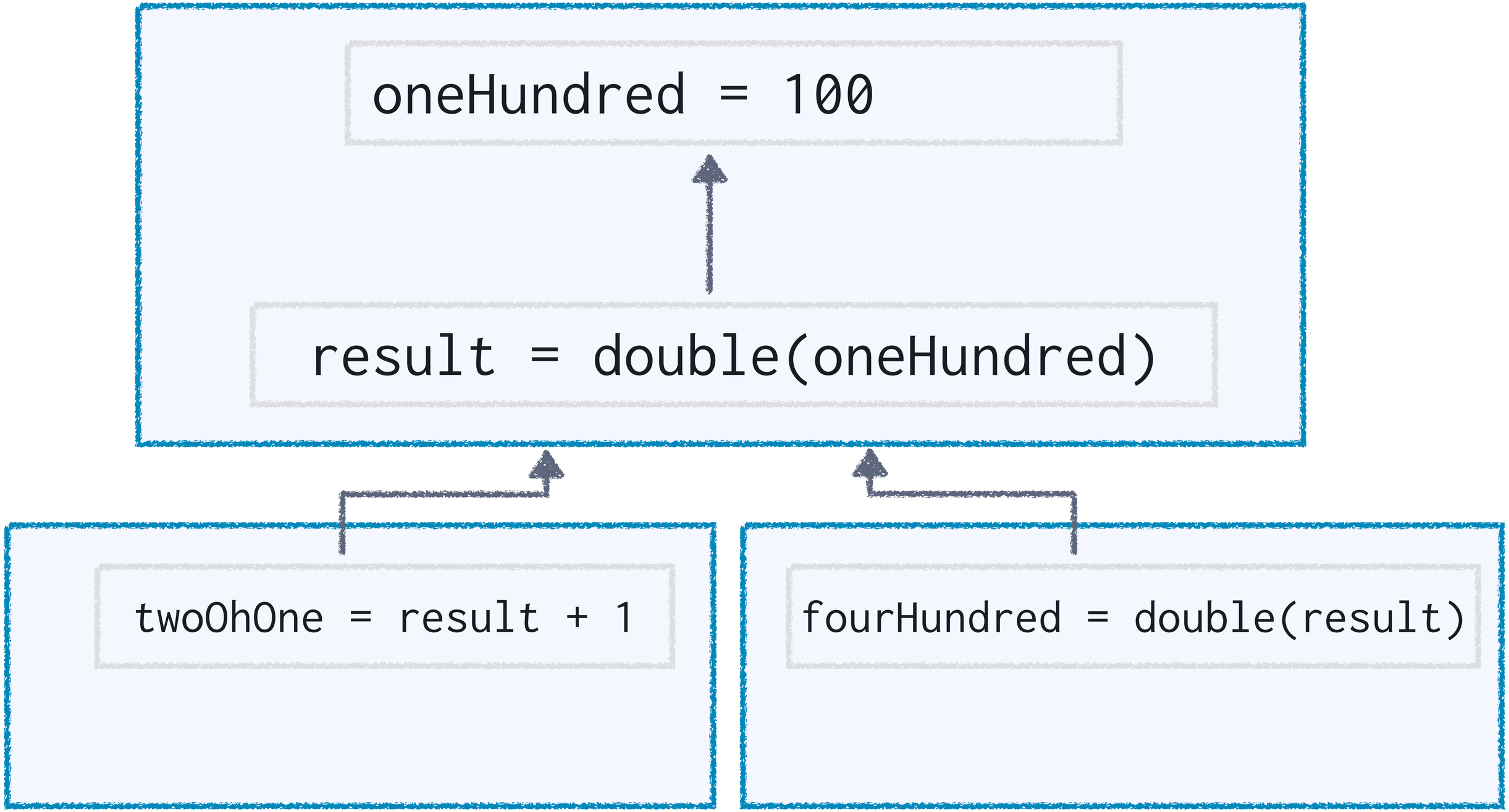
```
function double(x) {
  return x * 2;
}

var oneHundred = double(50);

var result = double(oneHundred);

var twoOhOne     = result + 1;
var fourHundred = double(result);
```

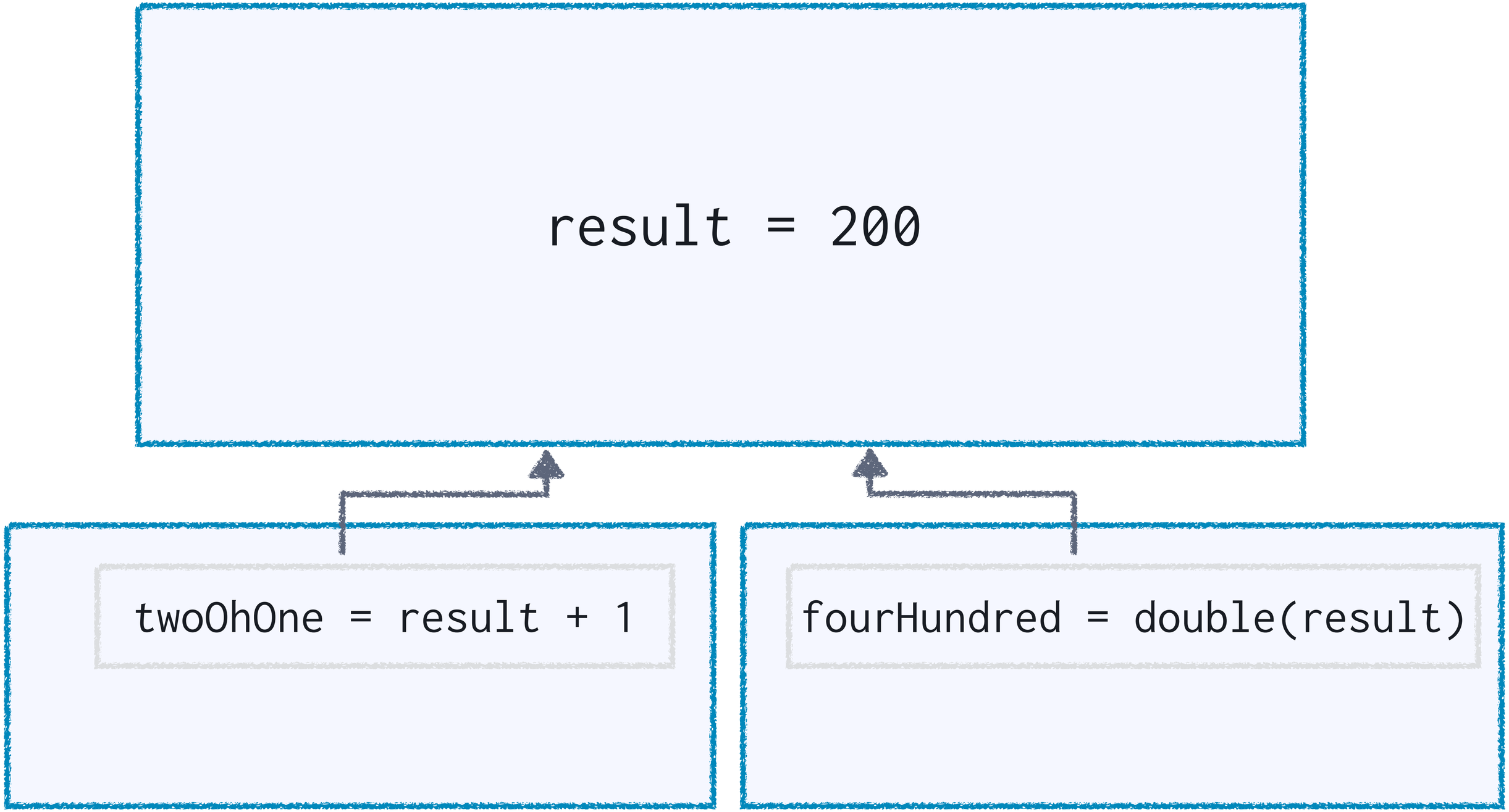


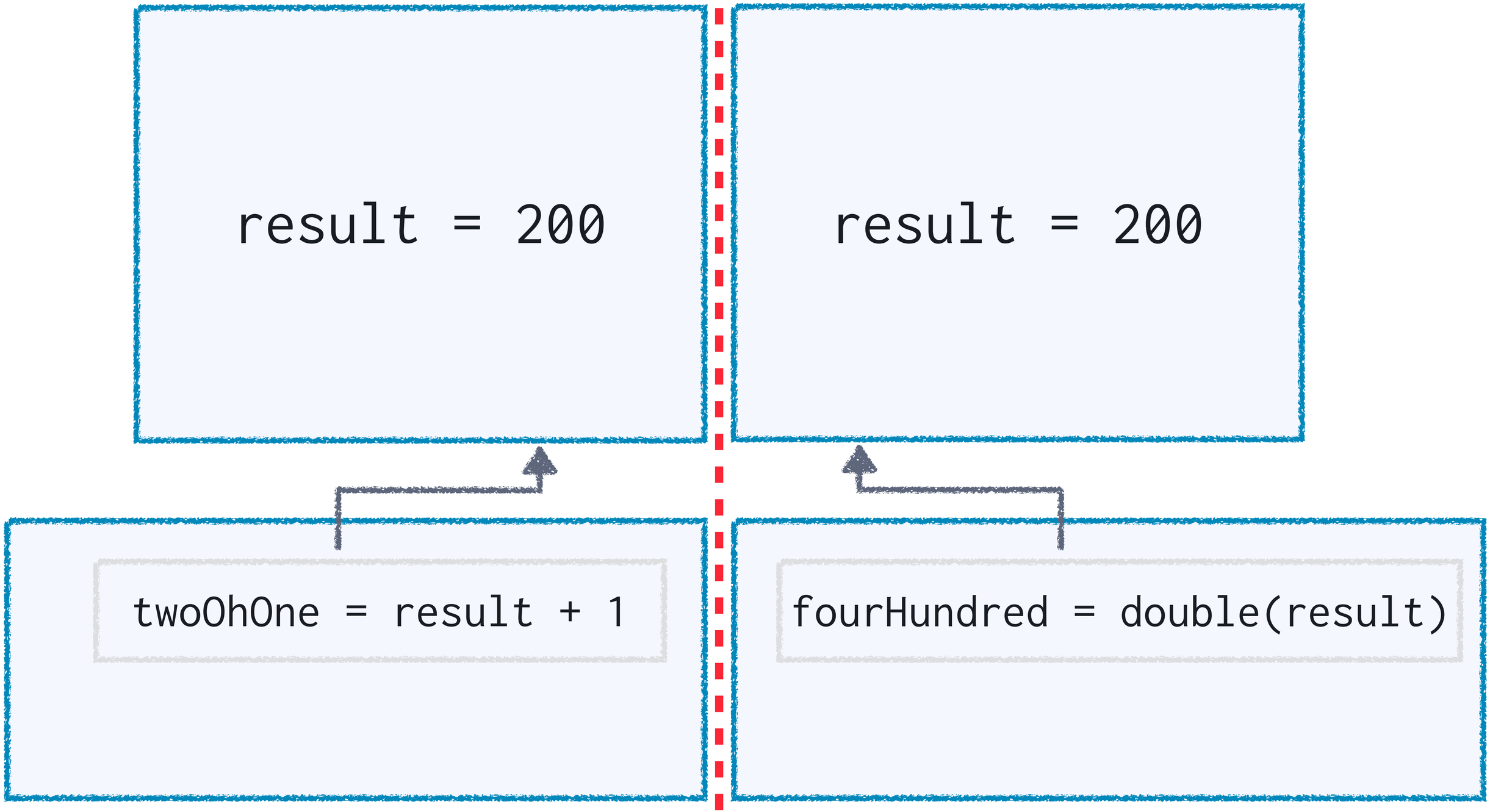

oneHundred = double(50)
result = double(oneHundred)
twoOhOne = result + 1
fourHundred = double(result)

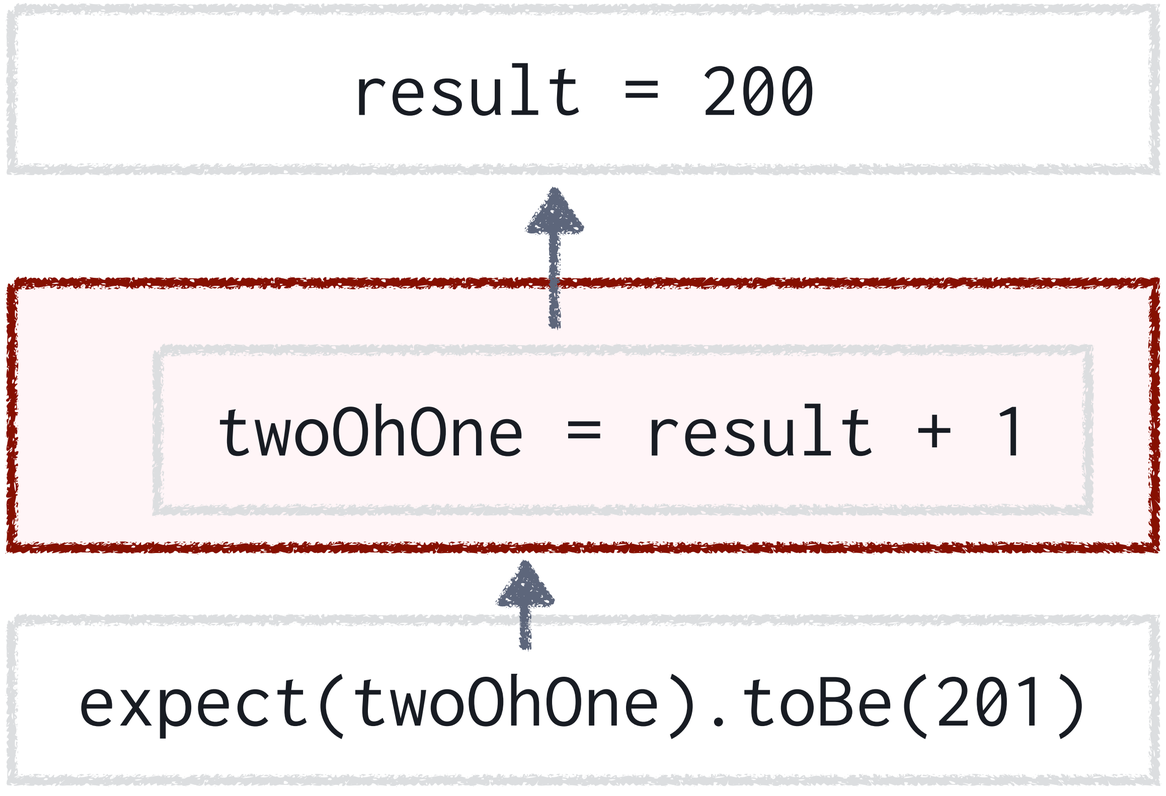

oneHundred = double(50)
<300 lines of code>
result = double(oneHundred)
twoOhOne = result + 1
fourHundred = double(result)

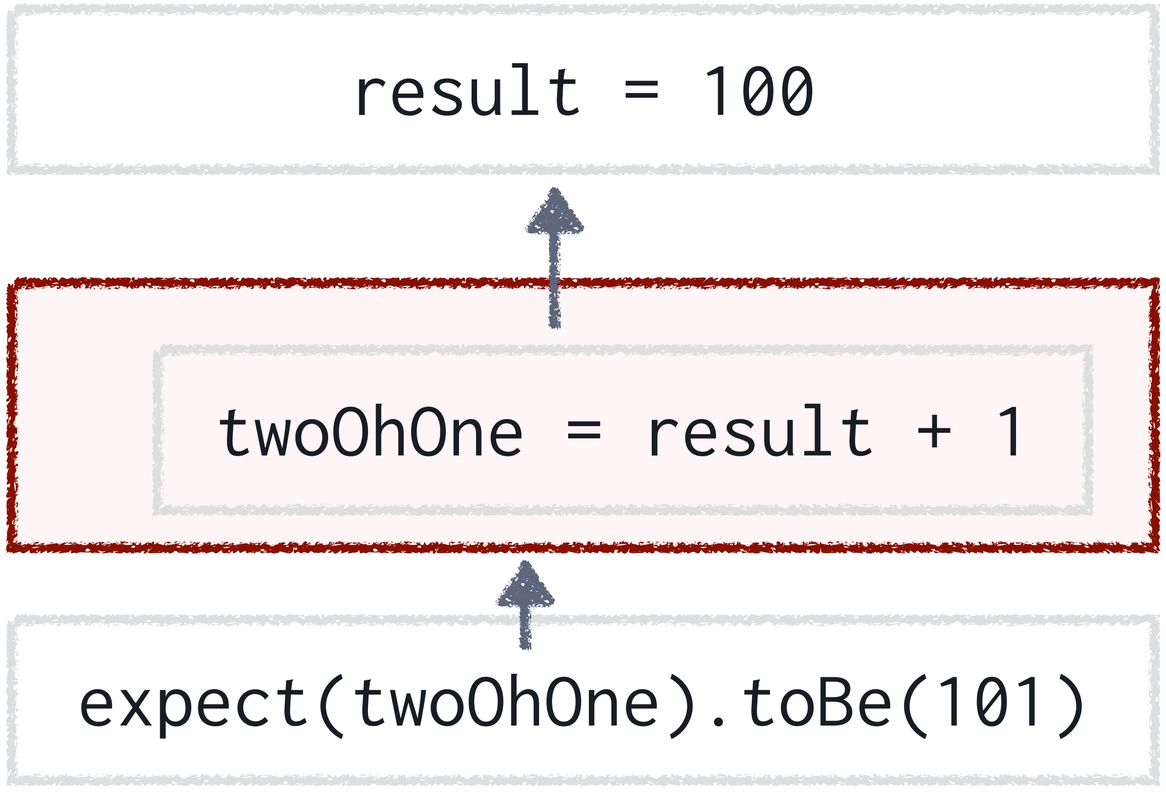

oneHundred = double(50)
result = double(oneHundred)
twoOhOne = result + 1
fourHundred = double(result)

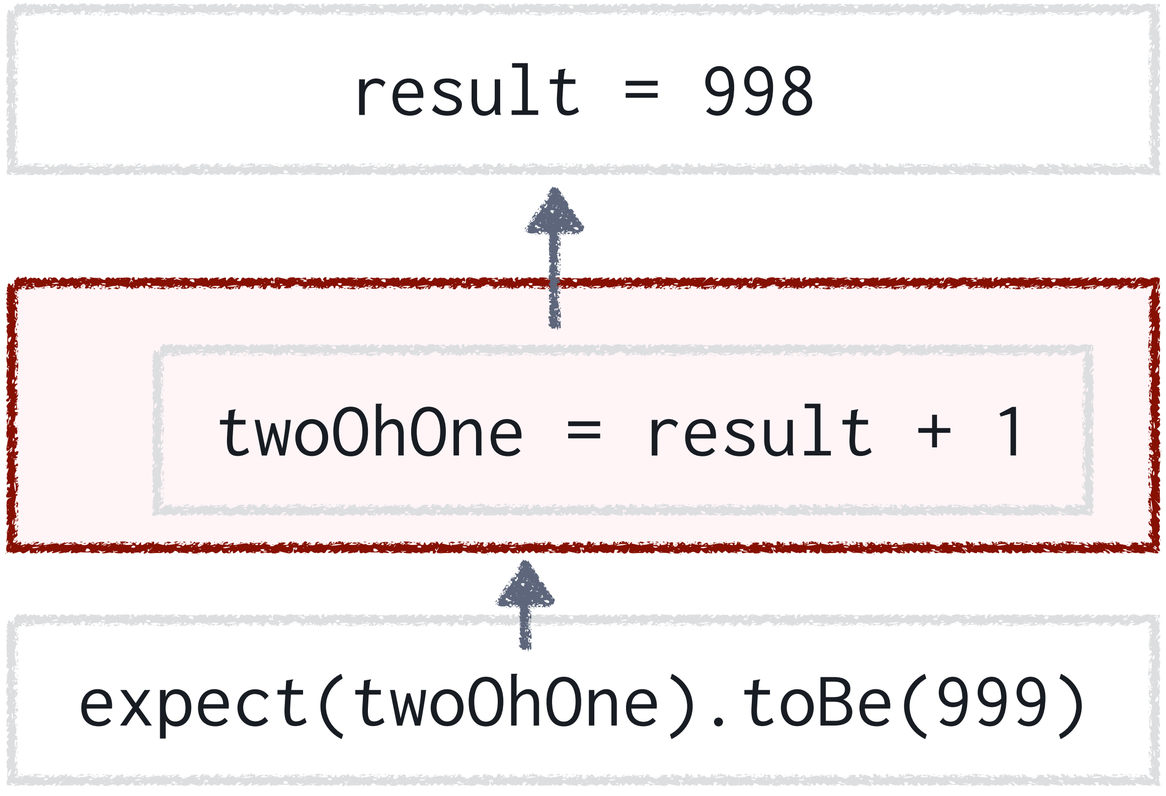

oneHundred = 100
result = double(oneHundred)
twoOhOne = result + 1
fourHundred = double(result)


result = 200
twoOhOne = result + 1
fourHundred = double(result)


result = 200
result = 200
twoOhOne = result + 1
fourHundred = double(result)


result = 200
twoOhOne = result + 1
expect(twoOhOne).toBe(201)


result = 100
twoOhOne = result + 1
expect(twoOhOne).toBe(101)


result = 998
twoOhOne = result + 1
expect(twoOhOne).toBe(999)

Modularity

Modularity

of evaluation aids:

Modularity

of evaluation aids:

Refactoring

Modularity

of evaluation aids:

Refactoring

Testing

Modularity

of evaluation aids:

Refactoring

Testing

Concurrency + Parallelism

Modularity

of evaluation aids:

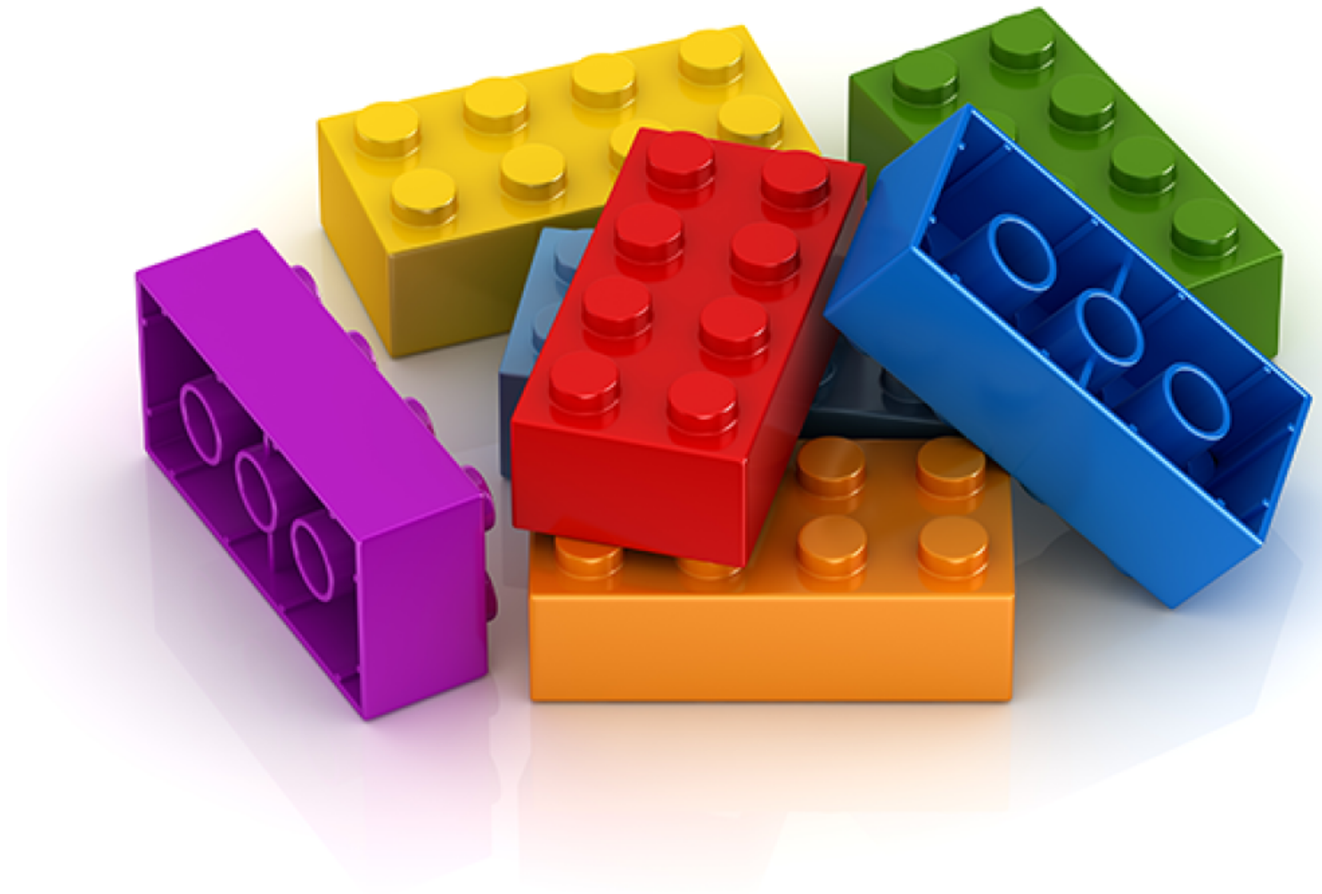
Refactoring

Testing

Concurrency + Parallelism

Composition + Decomposition

Composition



"But sometimes you take the blue 2x2 brick and the gray 4x1 brick and put them together in a certain way, and you realize, 'that's a useful piece that I need often'.

"So now you've come up with a new 'piece', a combination of two other pieces, and you can reach for that kind of piece now anytime you need it. It's more effective to recognize and use this compound blue-gray L-brick thing where it's needed than to separately think about assembling the two individual bricks each time."

— Kyle Simpson,
"Functional-Light JavaScript"

"Composition is combining things in a way that allows you to further combine them.

"You can't distinguish the combined thing from the atomic one."

— @kritzcreek

"ab" + "cd"

"abcd"

```
foo(bar(x))
```



```
function fooAndBar(x) {
  return foo(bar(x));
}
fooAndBar(x)
```

et al

Expressions

double(2)

```
double(2)
```

```
double(2)
```

```
double(2)
```

4

```
function double(x) {
  return x * 2;
}
```

```
function double(x=2) {
  return x * 2;
}
```

2 * 2

2 * 2

4

double(2)


```
double(2)
```

4

Values & Functions

Functions
are also Values

(And values can be represented with only functions.)

LAMBDA CALCULUS

(And [illegible] only functions)

(And values can be represented with only functions.)

Functions
are also Values

```
var plusOne = function(x) {
  return x + 1;
}
```

```
var plusOne = function(x) {
  return x + 1;
}
```

```
var plusOne = function(x) {
  return x + 1;
}

[1,2,3].map(plusOne);
```

```
var plusOne = function(x) {
  return x + 1;
}

[1,2,3].map(plusOne);
```

(That `map()` is a "Higher-Order Function"; it takes a function as an argument.)

a desire for

Referential Transparency

+

the helpful feature of

Functions as Values

+

abstracting out common behaviour with

Decomposition

… gives us …

```
map(x => …)
fold((a,x) => …,…)
chain(x => …)
```

```
[…].map(x => …)
[…].reduce((a,x) => …,…)
[…].flatMap(x => …)
```

```
[…].map(x => …)
[…].reduce((a,x) => …,…)
[…].flatMap(x => …)
[…].filter(x => …)
[…].find(x => …)
[…].reduceRight((x,xs)=>
```

eg.

```
function mysteryCode(thisYear, birthYears) {
  var i, total = 0;
  for (i = 0; i < birthYears.length; i++){
    var age = thisYear - birthYears[i];
    if (age > 20) {
      total += age;
    }
  }
  return total;
}

mysteryCode(2017, [1999, 1980, 1976]) // 78
```

```
function mysteryCode(thisYear, birthYears) {
  return birthYears
    .map(birthYr => thisYear - birthYr)
    .filter(age => age > 20)
    .reduce((total,age) => total + age, 0)
}
```

```
mysteryCode(2017, [1999, 1980, 1976]) // 78
```

```
function mysteryCode(thisYear, birthYears) {
  return birthYears
    .map(birthYr => thisYear - birthYr)
    .filter(age => age > 20)
    .reduce((total,age) => total + age, 0)
}
```

```
function mysteryCode(thisYear, birthYears) {
  var i, total = 0;
  for (i = 0; i < birthYears.length; i++){
    var age = thisYear - birthYears[i];
    if (age > 20) {
      total += age;
    }
  }
  return total;
}
```

```
function mysteryCode(thisYear, birthYears) {
  return birthYears
    .map(birthYr => thisYear - birthYr)
    .filter(age => age > 20)
    .reduce((total,age) => total + age, 0)
}
```

```
function mysteryCode(thisYear, birthYears) {
  var i, total = 0;
  for (i = 0; i < birthYears.length; i++){
    var age = thisYear - birthYears[i];
    if (age > 20) {
      total += age;
    }
  }
  return total;
}
```

```
function mysteryCode(thisYear, birthYears) {
  return birthYears
    .map(birthYr => thisYear - birthYr)
    .filter(age => age > 20)
    .reduce((total,age) => total + age, 0)
}
```

```
function mysteryCode(thisYear, birthYears) {
  var i, total = 0;
  for (i = 0; i < birthYears.length; i++){
    var age = thisYear - birthYears[i];
    if (age > 20) {
      total += age;
    }
  }
  return total;
}
```

```
function mysteryCode(thisYear, birthYears) {
  return birthYears
    .map(birthYr => thisYear - birthYr)
    .filter(age => age > 20)
    .reduce((total,age) => total + age, 0)
}
```

```
function mysteryCode(thisYear, birthYears) {
  var i, total = 0;
  for (i = 0; i < birthYears.length; i++){
    var age = thisYear - birthYears[i];
    if (age > 20) {
      total += age;
    }
  }
  return total;
}
```

```
function mysteryCode(thisYear, birthYears) {
  return birthYears
    .map(birthYr => thisYear - birthYr)
    .filter(age => age > 20)
    .reduce((total,age) => total + age, 0)
}
```

```
function mysteryCode(thisYear, birthYears) {
  var i, total = 0;
  for (i = 0; i < birthYears.length; i++){
    var age = thisYear - birthYears[i];
    if (age > 20) {
      total += age;
    }
  }
  return total;
}
```

```
function mysteryCode(thisYear, birthYears) {
  return birthYears
    .map(birthYr => thisYear - birthYr)
    .filter(age => age > 20)
    .reduce((total,age) => total + age, 0)
}
```

```
function mysteryCode(thisYear, birthYears) {
  var i, total = 0;
  for (i = 0; i < birthYears.length; i++){
    var age = thisYear - birthYears[i];
    if (age > 20) {
      total += age;
    }
  }
  return total;
}
```

```
function mysteryCode(thisYear, birthYears) {
  return birthYears
    .map(birthYr => thisYear - birthYr)
    .filter(age => age > 20)
    .reduce((total,age) => total + age, 0)
}
```

```
function mysteryCode(thisYear, birthYears) {
  var i, total = 0;
  for (i = 0; i < birthYears.length; i++){
    var age = thisYear - birthYears[i];
    if (age > 20) {
      total += age;
    }
  }
  return total;
}
```

```
function mysteryCode(thisYear, birthYears) {
  return birthYears
    .map(birthYr => thisYear - birthYr)
    .filter(age => age > 20)
    .reduce((total,age) => total + age, 0)
}
```

```
function mysteryCode(thisYear, birthYears) {
  var i, total = 0;
  for (i = 0; i < birthYears.length; i++){
    var age = thisYear - birthYears[i];
    if (age > 20) {
      total += age;
    }
  }
  return total;
}
```

```
function mysteryCode(thisYear, birthYears) {
  return birthYears
    .map(birthYr => thisYear - birthYr)
    .filter(age => age > 20)
    .reduce((total,age) => total + age, 0)
}
```

```
function mysteryCode(thisYear, birthYears) {
  var i, total = 0;
  for (i = 0; i < birthYears.length; i++){
    var age = thisYear - birthYears[i];
    if (age > 20) {
      total += age;
    }
  }
  return total;
}
```

```
def mysteryCode(thisYear, birthYears)
  birthYears
    .lazy
    .map {|birthYr| thisYear - birthYr }
    .select {|age| age > 20 }
    .reduce(0) {|total,age| total + age }
end
```

```
def mysteryCode(thisYear, birthYears)
  birthYears
    .lazy
    .map {|birthYr| thisYear - birthYr }
    .select {|age| age > 20 }
    .reduce(0) {|total,age| total + age }
end
```

"Expression-Oriented" Languages

```
main =
  print(toString(length(usersWith(hackers, "e"))))

usersWith users letter =
  filter(
    (u => contains(letter, u)),
    users
  )

hackers =
  [ "pjf", "zer0c00l", "crash_override" ]
```

```
main =
  print(toString(length(usersWith(hackers, "e"))))

usersWith users letter =
  filter(
    (u => contains(letter, u)),
    users
  )

hackers =
  [ "pjf", "zer0c00l", "crash_override" ]
```

```
main =
  print(toString(length(
    filter(
      (u => contains("e", u)),
      hackers
    )
  )))

hackers =
  [ "pjf", "zer0c00l", "crash_override" ]
```

```
main =
  print(toString(length(
    filter(
      (u => contains("e", u)),
      hackers
    )
  )))

hackers =
  [ "pjf", "zer0c00l", "crash_override" ]
```

```
main =
  print(toString(length(
    filter(
      (u => contains("e", u)),
      hackers
    )
  )))

hackers =
  [ "pjf", "zer0c00l", "crash_override" ]
```

```
main =
  print(toString(length(
    filter(
      (u => contains("e", u)),
      [ "pjf", "zer0c00l", "crash_override" ]
    )
  )))
```

```
main =
  print(toString(length(
    filter(
      (u => contains("e", u)),
      [ "pjf", "zer0c00l", "crash_override" ]
    )
  )))
```

```
main =
  print(toString(length(
    ["zer0c00l", "crash_override" ]
  )))
```

```
main =
  print(toString(
    2
  ))
```

```
main =
  print(
   "2"
  )
```

Language Affordances

```
[1980, 1998, 2006]
  .map(yr => now - yr)
  .filter(age => age > 20)
  .reduce((t,a) => t+a, 0)
```

```
reduce((t,a) => t + a, 0,
  filter(age => age > 20,
    map(yr => now - yr,
      [1980, 1998, 2006]
    )))
```

```
[1980, 1998, 2006]
  |> map (\yr -> now - yr)
  |> filter (\age -> age > 20)
  |> foldl (\t a -> t + a) 0
```

“ ”

"Faux-O"

— Gary Bernhardt, "Boundaries"
https://www.youtube.com/watch?v=yTkzNHF6rMs

```
class MyArray {
  constructor(v){ this.items = v;}

  map(func) {
    var i, newItems = [];
    for (i=0; i < this.items.length; i++) {
      newItems.push(  func(this.items[i])  );
    }
    return new MyArray(newItems);
  }

  // ...
}
```

```
class MyArray {
  constructor(v){ this.items = v;}

  map(func) {
    var i, newItems = [];
    for (i=0; i < this.items.length; i++) {
      newItems.push(  func(this.items[i])  );
    }
    return new MyArray(newItems);
  }

  // ...
}
```

```
class MyArray {
  constructor(v){ this.items = v;}

  map(func) {
    var i, newItems = [];
    for (i=0; i < this.items.length; i++) {
      newItems.push(  func(this.items[i])  );
    }
    return new MyArray(newItems);
  }

  // ...
}
```

```
class MyArray {
  constructor(v){ this.items = v;}

  map(func) {
    var i, newItems = [];
    for (i=0; i < this.items.length; i++) {
      newItems.push(  func(this.items[i])  );
    }
    return new MyArray(newItems);
  }

  // ...
}
```

```
class MyArray {
  constructor(v){ this.items = v;}

  map(func) {
    var i, newItems = [];
    for (i=0; i < this.items.length; i++) {
      newItems.push(  func(this.items[i])  );
    }
    return new MyArray(newItems);
  }

  // ...
}
```

```
class MyArray {
  constructor(v){ this.items = v;}

  map(func) {
    var i, newItems = [];
    for (i=0; i < this.items.length; i++) {
      newItems.push(  func(this.items[i])  );
    }
    return new MyArray(newItems);
  }

  // ...
}
```

```
var items = new MyArray([1,2,3]);


var r1 =
  items.map(x => x + 1)
    .map(x => x * 2);

var r2 =
  items.map(x => x + 1)
    .map(x => x * 2);
```

```
var items = new MyArray([1,2,3]);


var r1 =
  items.map(x => x + 1)
    .map(x => x * 2);

var r2 =
  items.map(x => x + 1)
    .map(x => x * 2);
```

```
var items = new MyArray([1,2,3]);
var plusOne = items.map(x => x + 1)

var r1 =
  plusOne
    .map(x => x * 2);

var r2 =
  plusOne
    .map(x => x * 2);
```

```
var items = new MyArray([1,2,3]);
var plusOne = items.map(x => x + 1)

var r1 =
  plusOne
    .map(x => x * 2);

var r2 =
  plusOne
    .map(x => x * 2);
```

Sanctuary

(sanctuary.js.org)

Folktale

(folktale.origamitower.com)

Ramda

(ramdajs.com)

Immutability

```
var evens =
  evenNumbers(bigList(12));
  // [2,4,6,8,10,12]

var tens =
  endsInZero(bigList(12));
  // [10]
```

```
var evens =
  evenNumbers(bigList(12));
  // [2,4,6,8,10,12]

var tens =
  endsInZero(bigList(12));
  // [10]
```

```
var list = bigList(12);

var evens =
  evenNumbers(list);
  // [2,4,6,8,10,12]


var tens =
  endsInZero(list);
  // []
```

```
var list = bigList(12);
var evens =
  evenNumbers(list);
  // [2,4,6,8,10,12]

var tens =
  endsInZero(list);
  // []💦
```

```
function evenNumbers(list) {
  var result = []
  while (list.length > 0) {
    var num = list.shift();
    // Changes the array!
    if (num % 2 == 0) {
      result.push(num)
    }
  }
  return result;
}
```

```
function evenNumbers(list) {
  var result = []
  while (list.length > 0) {
    var num = list.shift();
    // Changes the array!
    if (num % 2 == 0) {
      result.push(num)
    }
  }
  return result;
}
```

```
var list =
  bigList(12);
var evens =
  evenNumbers(list);
  // [2,4,6,8,10,12]
var tens =
  endsInZero(list);
  // []
```

Imagine a world where this... doesn't happen.

through

Quality/Property of the Code

(via Referential Transparency)

or

Language Restriction

or

API Restriction

```
var Immutable =
  require('seamless-immutable');
var list =
  Immutable(bigList(12));

var evens =
  evenNumbers(list);
  // ...
var tens =
  endsInZero(list);
  // ...
```

```
var Immutable =
  require('seamless-immutable');
var list =
  Immutable(bigList(12));

var evens =
  evenNumbers(list);
  // 💥 ImmutableError: The
  // shift method cannot be
  // invoked on an Immutable
  // data structure.
```

```
var list =
  Object.freeze(bigList(12));

var evens =
  evenNumbers(list);
  // 💥 TypeError: Cannot
  // add/remove sealed array
  // elements
```

```
var list =
  Object.freeze([{a:123},{a:456}]);

list[0].a = 999;
console.log(list[0].a);
// 999
```

seamless-immutable

Immutable.js

Effects as Values

Representing impure actions as values can make them pure.

Representing impure actions as instructions to be later interpreted can make them pure.

Representing impure actions as instructions to be later interpreted can make them pure.

1.

```
function makeMessage(msgText) {
  return { text: msgText };
}

function evaluateMessage(msg) {
  console.log(msg.text);
}

const hi = makeMessage("Hello");
// => { text="hello" }
```

1.

```
function makeMessage(msgText) { … }
function evaluateMessage(msg) { … }
const hi = makeMessage("Hello");
// => { text="hello" }
```

1.

```
function makeMessage(msgText) { … }
function evaluateMessage(msg) { … }
const hi = makeMessage("Hello");
// => { text="hello" }

const messages = [ hi, hi, hi, hi ]
// => [ { text="hello" },
        { text="hello" },
        { text="hello" },
        { text="hello" } ]
```

1.

```
function makeMessage(msgText) { … }
function evaluateMessage(msg) { … }
const hi = makeMessage("Hello");
const messages = [ hi, hi, hi, hi ];
// => [ { text="hello" }, ... ]
```

1.

```
function makeMessage(msgText) { … }
function evaluateMessage(msg) { … }
const hi = makeMessage("Hello");
const messages = [ hi, hi, hi, hi ];
// => [ { text="hello" }, ... ]



messages.forEach(function(msg){
  evaluateMessage(msg);
}); // hello↵hello↵hello↵hello↵
```

Pure

```
function makeMessage(msgText) { … }
function evaluateMessage(msg) { … }
const hi = makeMessage("Hello");
const messages = [ hi, hi, hi, hi ];
// => [ { text="hello" }, ... ]
```

```
messages.forEach(function(msg){
  evaluateMessage(msg);
}); // hello↵hello↵hello↵hello↵
```

Pure

```
function makeMessage(msgText) { … }
function evaluateMessage(msg) { … }
const hi = makeMessage("Hello");
const messages = [ hi, hi, hi, hi ];
// => [ { text="hello" }, ... ]
```

```
messages.forEach(function(msg){
  evaluateMessage(msg);
}); // hello↵hello↵hello↵hello↵
```

Impure

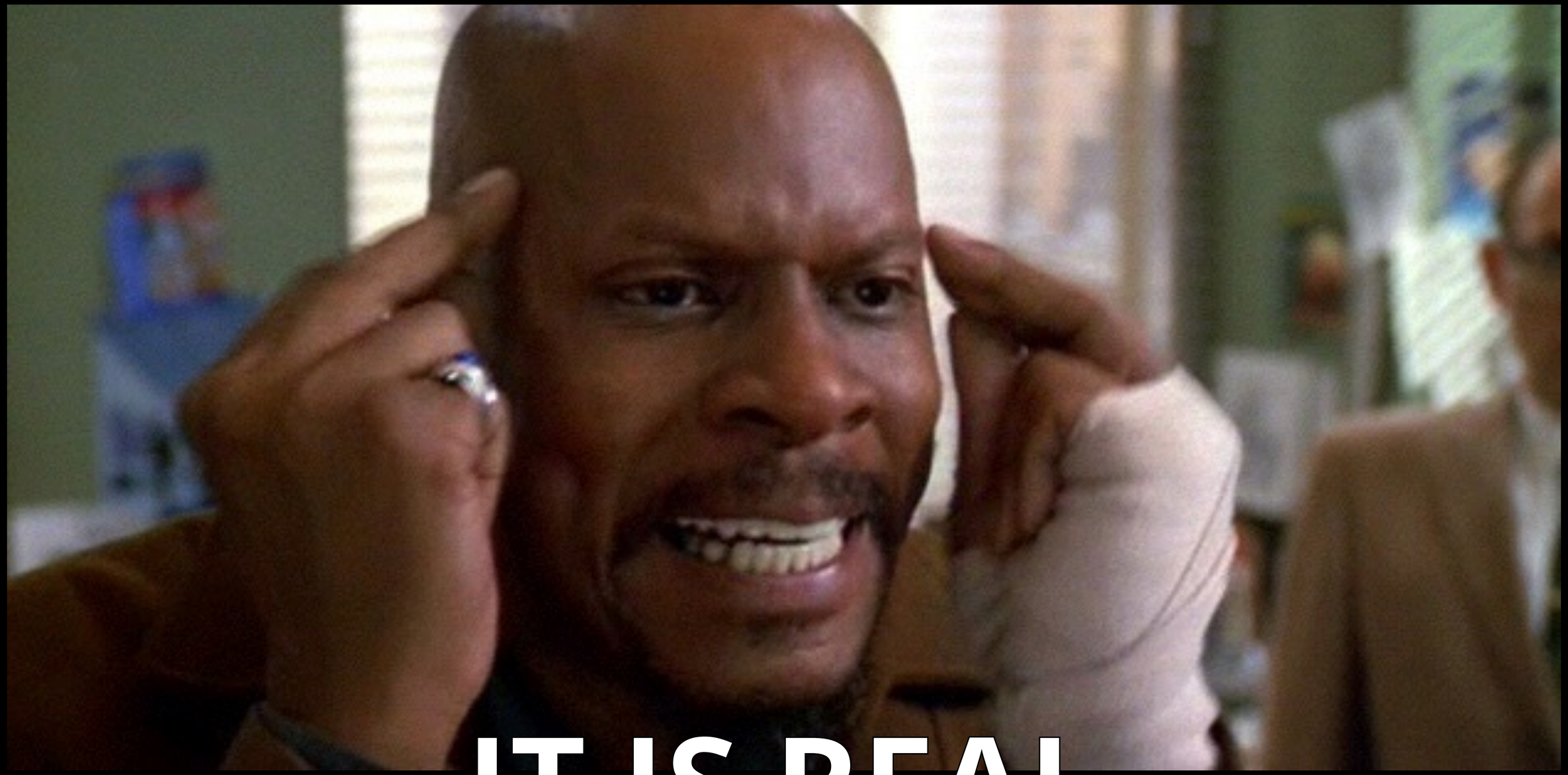
Pure

```
function makeMessage(msgText) { … }
function evaluateMessage(msg) { … }
const hi = makeMessage("Hello");
const messages = [ hi, hi, hi, hi ];
// => [ { text="hello" }, ... ]
```

```
messages.forEach(function(msg){
  evaluateMessage(msg);
}); // hello↵hello↵hello↵hello↵
```

Impure

I am aware this seems ridiculous.


IT IS REAL

```
const eventualHtml =
  futureFetch("http://example.com/blah")
```

```
const eventualHtml =
  futureFetch("http://example.com/blah")
    .map(r => parseHtml(r.text())
```

```
const eventualHtml =
  futureFetch("http://example.com/blah")
    .map(r => parseHtml(r.text())
const linkedPages =
  eventualHtml.chain(h =>
    scrapeUrls(h).parallel(3, futureFetch)
  )
```

Pure

```
const eventualHtml =
  futureFetch("http://example.com/blah")
    .map(r => parseHtml(r.text())

const linkedPages =
  eventualHtml.chain(h =>
    scrapeUrls(h).parallel(3, futureFetch)
  )
```

Pure

```
const eventualHtml =
  futureFetch("http://example.com/blah")
    .map(r => parseHtml(r.text())

const linkedPages =
  eventualHtml.chain(h =>
    scrapeUrls(h).parallel(3, futureFetch)
  )
```

Pure

```
const eventualHtml =
  futureFetch("http://example.com/blah")
    .map(r => parseHtml(r.text())

const linkedPages =
  eventualHtml.chain(h =>
    scrapeUrls(h).parallel(3, futureFetch)
  )
```

```
linkedPages.fork(
  e => { throw e; },
  pages => saveToDb(pages)
)
```

```
const eventualHtml =
  futureFetch("http://example.com/blah")
    .map(r => parseHtml(r.text())

const linkedPages =
  eventualHtml.chain(h =>
    scrapeUrls(h).parallel(3, futureFetch)
  )
```

Pure

```
linkedPages.fork(
  e => { throw e; },
  pages => saveToDb(pages)
)
```

Impure

```
const eventualHtml =
  futureFetch("http://example.com/blah")
    .map(r => parseHtml(r.text())

const linkedPages =
  eventualHtml.chain(h =>
    scrapeUrls(h).parallel(3, futureFetch)
  )
```

Pure

```
linkedPages.fork(
  e => { throw e; },
  pages => saveToDb(pages)
)
```

Impure

Pure

```
const eventualHtml =
  futureFetch("http://example.com/blah")
    .map(r => parseHtml(r.text()))

const linkedPages =
  eventualHtml.chain(h =>
    scrapeUrls(h).parallel(3, futureFetch)
  )
```

```
linkedPages.fork(
  e => { throw e; },
  pages => saveToDb(pages)
)
```

Impure

```
const eventualHtml =
  futureFetch("http://example.com/blah")
    .map(r => parseHtml(r.text())

const linkedPages =
  eventualHtml.chain(h =>
    scrapeUrls(h).parallel(3, futureFetch)
  )
```

Pure

```
linkedPages.fork(
  e => { throw e; },
  pages => saveToDb(pages)
)
```

Impure

```
const eventualHtml =
  futureFetch("http://example.com/blah")
    .map(r => parseHtml(r.text())

const linkedPages =
  eventualHtml.chain(h =>
    scrapeUrls(h).parallel(3, futureFetch)
  )
```

Pure

```
linkedPages.fork(
  e => { throw e; },
  pages => saveToDb(pages)
)
```

Impure

Pure

```
const eventualHtml =
  futureFetch("http://example.com/blah")
    .map(r => parseHtml(r.text())
const linkedPages =
  eventualHtml.chain(h => ...);
const linkedImages =
  eventualHtml.chain(h => ...);
```

```
linkedPages.fork(...);
linkedImages.fork(...);
```

Impure

```
const eventualHtml =
  futureFetch("http://example.com/blah")
    .map(r => parseHtml(r.text()))
const linkedPages =
  eventualHtml.chain(h => ...);
const linkedImages =
  eventualHtml.chain(h => ...);
```

Pure

```
linkedPages.fork(...);
linkedImages.fork(...);
```

Impure

for this very specific example:

Fluture

data.task (Folktale)

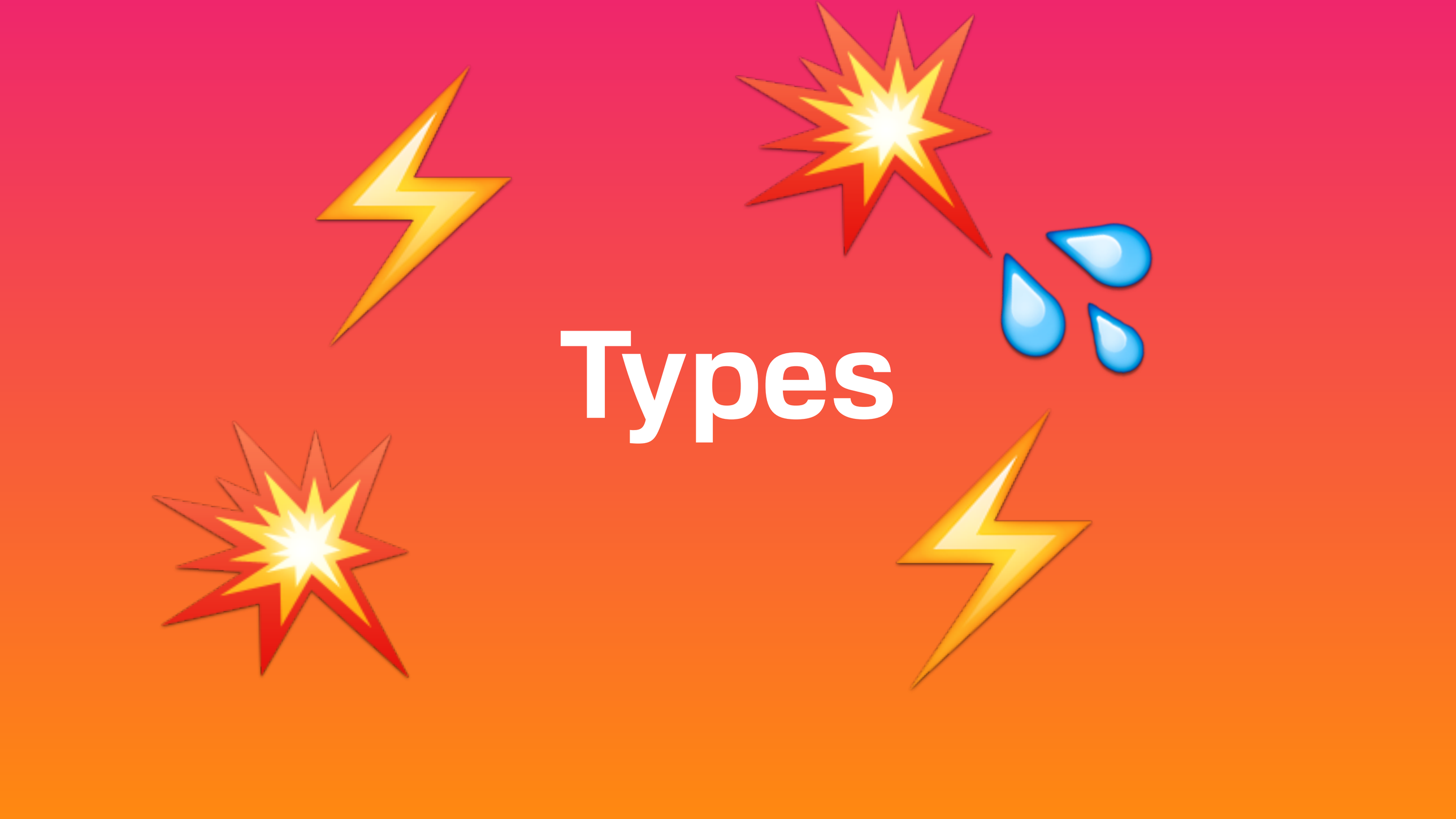
Types

Types

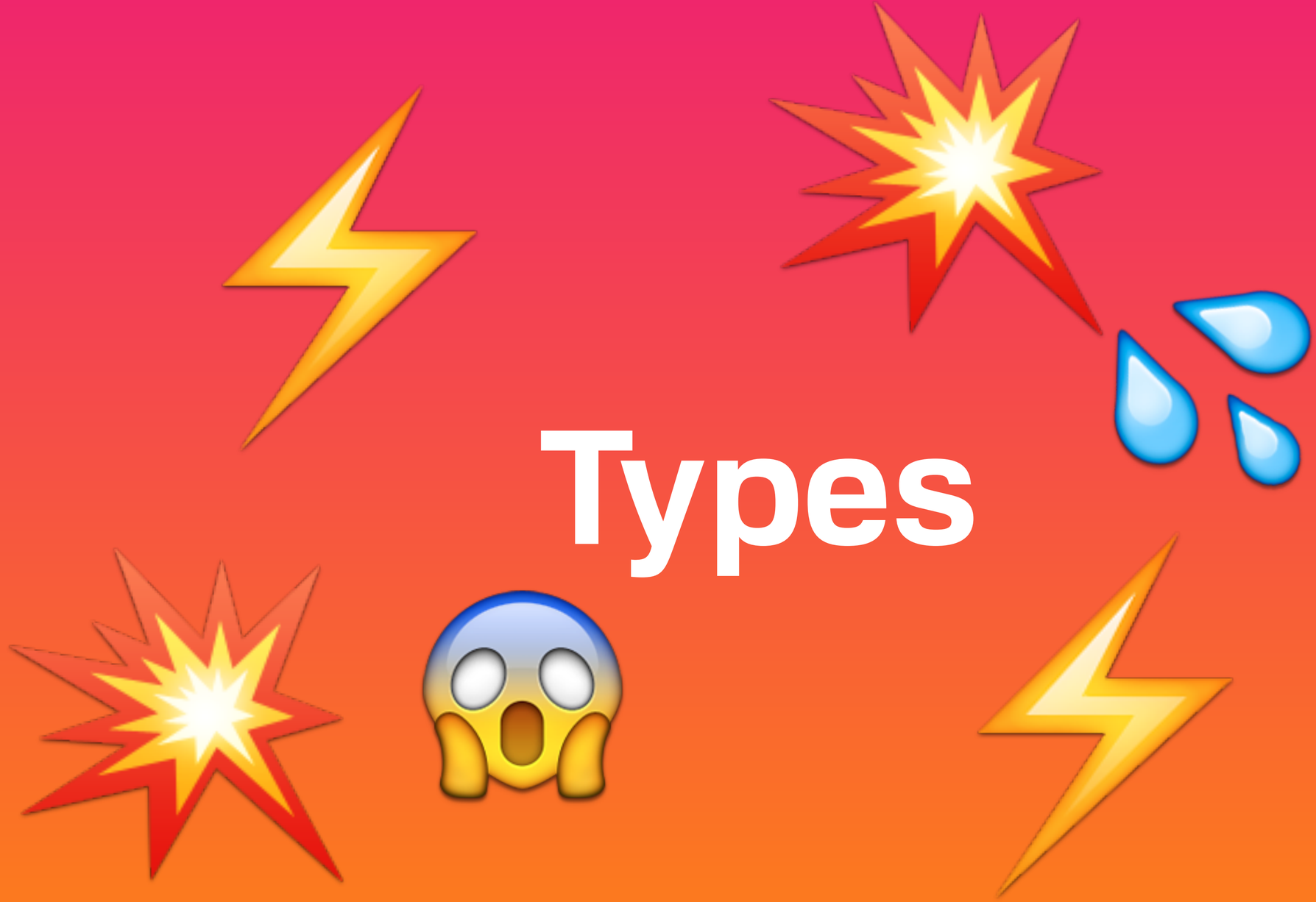
Types

Types

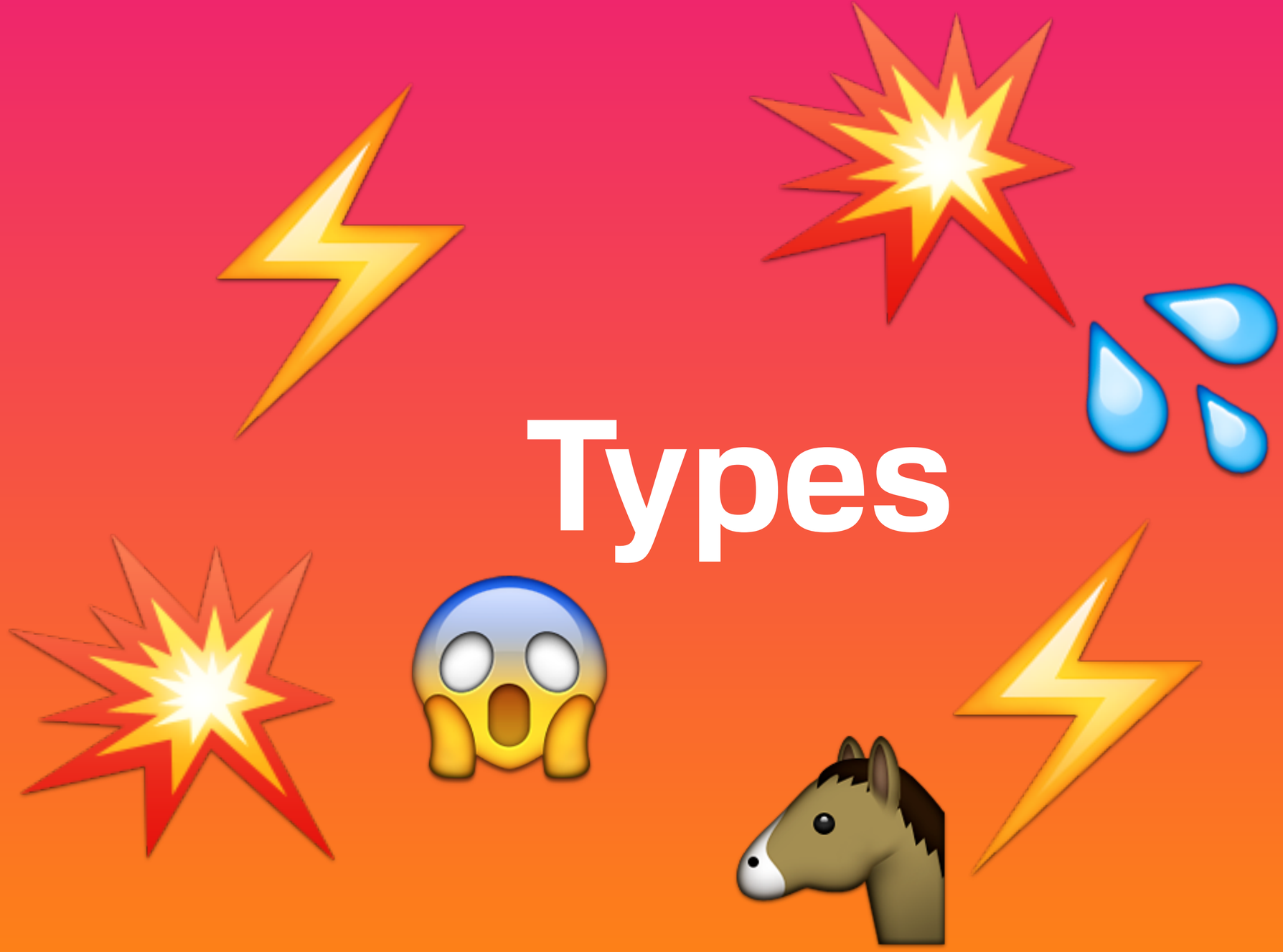
Types

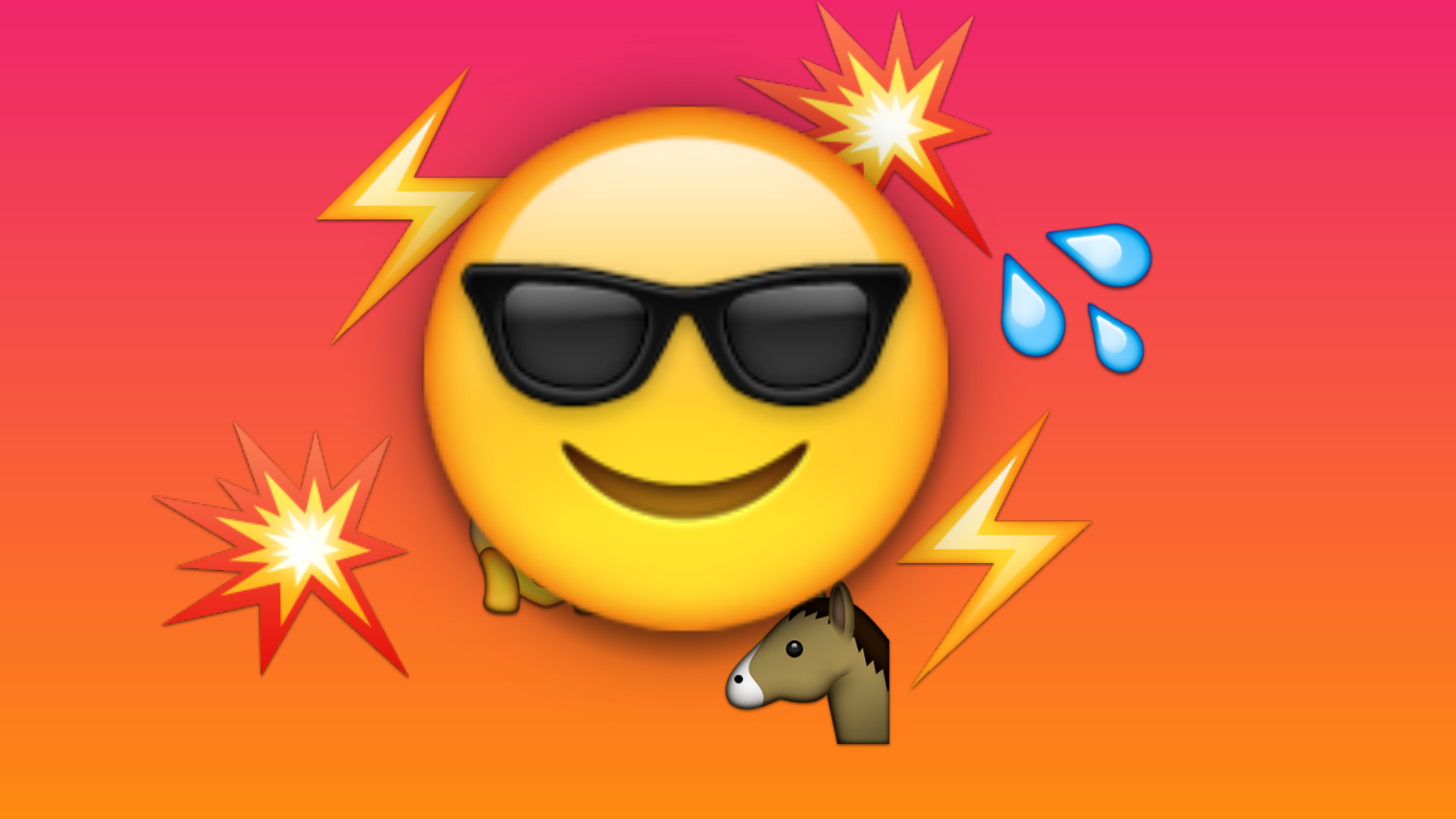

Types

Types



Types





Types

```
var backwards =
  myUtils.reverseStr(
    undefined
  );
```

```
var backwards =
  myUtils.reverseStr(
    undefined
  );
// TypeError: Cannot read
// property 'split' of undefined
//     at reverse (...)
```

```
var backwards =
  myUtils.reverseStr(
    7
  );
```

```
var backwards =
  myUtils.reverseStr(
    7
  );
// TypeError: x.split is not a function
//    at reverse (...)
```

```
function (x) {
  var backwards =
    myUtils.reverseStr(
      x
    );
  // ...
}
```

```
function (x) {
  var backwards =
    myUtils.reverseStr(
      x
    );
  // ...
}
// ...?
```

```
function reverseStr(x) {
  return x.split("")
          .reverse()
          .join("");
}
```

```
function reverseStr(x) {
  return x.split("")
          .reverse()
          .join("");
}
```

```
function reverseStr(
  x: string
): string {
  return x.split("")
          .reverse()
          .join("");
}
```

```
function reverseStr(
  x: string
): string {
  return x.split("")
          .reverse()
          .join("");
}
reverseStr("hello"); // 👍
```

```
function reverseStr(
  x: string
): string {
  return x.split("")
          .reverse()
          .join("");
}
reverseStr(7);          // 💥
```

```
function reverseStr(
  x: string
): string {
  return x.split("")
          .reverse()
          .join("");
}
reverseStr([1,2,3]); // 💥
```

```
$ cat example.js
```

```
$ cat example.js
// @flow
function reverseStr(x: string): string {
  return x.split("").reverse().join("");
}
console.log(reverseStr("hello"));
```

```
$ cat example.js
// @flow
function reverseStr(x: string): string {
  return x.split("").reverse().join("");
}
console.log(reverseStr("hello"));
```

```
$ flow check
```

```
$ cat example.js
// @flow
function reverseStr(x: string): string {
  return x.split("").reverse().join("");
}
console.log(reverseStr("hello"));

$ flow check
Found 0 errors
```

```
$ cat example.js
// @flow
function reverseStr(x: string): string {
  return x.split("").reverse().join("");
}
console.log(reverseStr("hello"));

$ flow check
Found 0 errors

$ babel-node example.js
```

```
$ cat example.js
// @flow
function reverseStr(x: string): string {
  return x.split("").reverse().join("");
}
console.log(reverseStr("hello"));

$ flow check
Found 0 errors

$ babel-node example.js
olleh
```

```
$ cat example.js
// @flow
function reverseStr(x: string): string {
  return x.split("").reverse().join("");
}
console.log(reverseStr(987));
```

```
$ cat example.js
// @flow
function reverseStr(x: string): string {
  return x.split("").reverse().join("");
}
console.log(reverseStr(987));

$ flow check
example.js:5
  5: console.log(reverseStr(987));
                 ^^^^^^^^^^^^^^^^ function call
  5: console.log(reverseStr(987));
                            ^^^ number.
     This type is incompatible with
  2: function reverseStr(x: string): string {
                            ^^^^^^ string

Found 1 error
```

```
function reverseStr(
  x: string
): string {
  return x.split("")
          .reverse()
          .join("");
}
```

```
function sort(
  x: Array<number>
): Array<number> {
  return ...
}
```

```
function sort(
  x: Array<number>
): Array<number> {
  return ...
}


sort([3,7,1]);  // 👍
```

```
function sort(
  x: Array<number>
): Array<number> {
  return ...
}


sort("hi");      // 💥
```

```
function reverse<T>(
  x: Array<T>
): Array<T> {
  return ...
}
```

```
function reverse<T>(
  x: Array<T>
): Array<T> {
  return ...
}


reverse([1,2,3]);     // 👍
```

```
function reverse<T>(
  x: Array<T>
): Array<T> {
  return ...
}


reverse(['a','b']);  // 👍
```

```
function reverse<T>(
  x: Array<T>
): Array<T> {
  return ...
}


reverse([true]);      // 👍
```

```
function greet(
  x: {username: string}
): string {
  return "Hi, " + x.username;
}

greet(
 {username: "Jean", id: 24601}
); // 👍
```

```
function defaultTo(
  d: string, val?: string
): string {
  if (val === undefined)
    return d;
  else
    return val;
}

defaultTo("brave", adjective); // 👍
```

```
type Thing = boolean | number;
function thingToString(
  x: Thing
): string {
  return ...
}
```

```
type Action =
    { type: "LOGGED_IN", user: string }
  | { type: "LOGGED_OUT" }

function user(
  state: State, action: Action
): State {
  if (action.type === "LOGGED_IN") {
    return {name: action.user, isLoggedIn: true}
  }
  if (action.type === "LOGGED_OUT")
  ...
}
```

```
type Action =
    { type: "LOGGED_IN", user: string }
  | { type: "LOGGED_OUT" }

function user(
  state: State, action: Action
): State {
  if (action.type === "LOGGED_IN") {
    return {name: action.user, isLoggedIn: true}
  }
  if (action.type === "LOGGED_OUT")
  ...
}
```

```
type Action =
    { type: "LOGGED_IN", user: string }
  | { type: "LOGGED_OUT" }

function user(
  state: State, action: Action
): State {
  if (action.type === "LOGGED_IN") {
    return {name: action.user, isLoggedIn: true}
  }
  if (action.type === "LOGGED_OUT")
  ...
}
```

```
type Action =
    { type: "LOGGED_IN", user: string }
  | { type: "LOGGED_OUT" }

function user(
  state: State, action: Action
): State {
  if (action.type === "LOGGED_IN") {
    return {name: action.user, isLoggedIn: true}
  }
  if (action.type === "LOGGED_OUT")
  ...
}
```

```
type Action =
    { type: "LOGGED_IN", user: string }
  | { type: "LOGGED_OUT" }

function user(
  state: State, action: Action
): State {
  if (action.type === "LOGGED_IN") {
    return {name: action.user, isLoggedIn: true}
  }
  if (action.type === "LOGGED_OUT")
  ...
}
```

```
function reverseStr(
  x: string
): string {
  return x.split("")
          .reverse()
          .join("");
}
```

```
function reverseStr(x) {
  return x.split("")
          .reverse()
          .join("");
}
```

Flow
TypeScript

… and Google Closure (?)

So...

Expressions

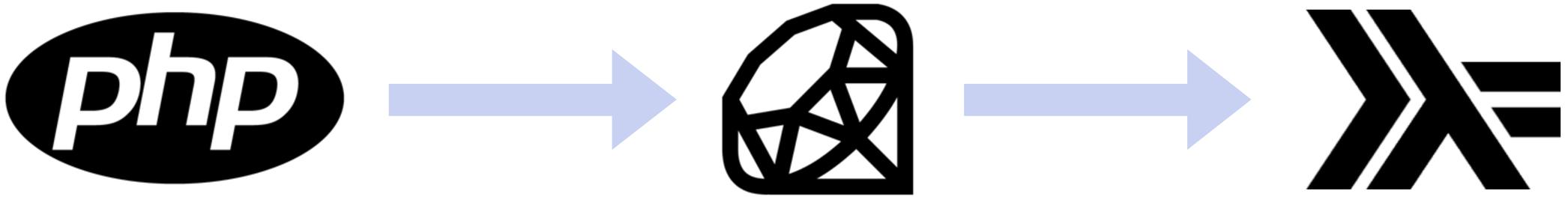

php

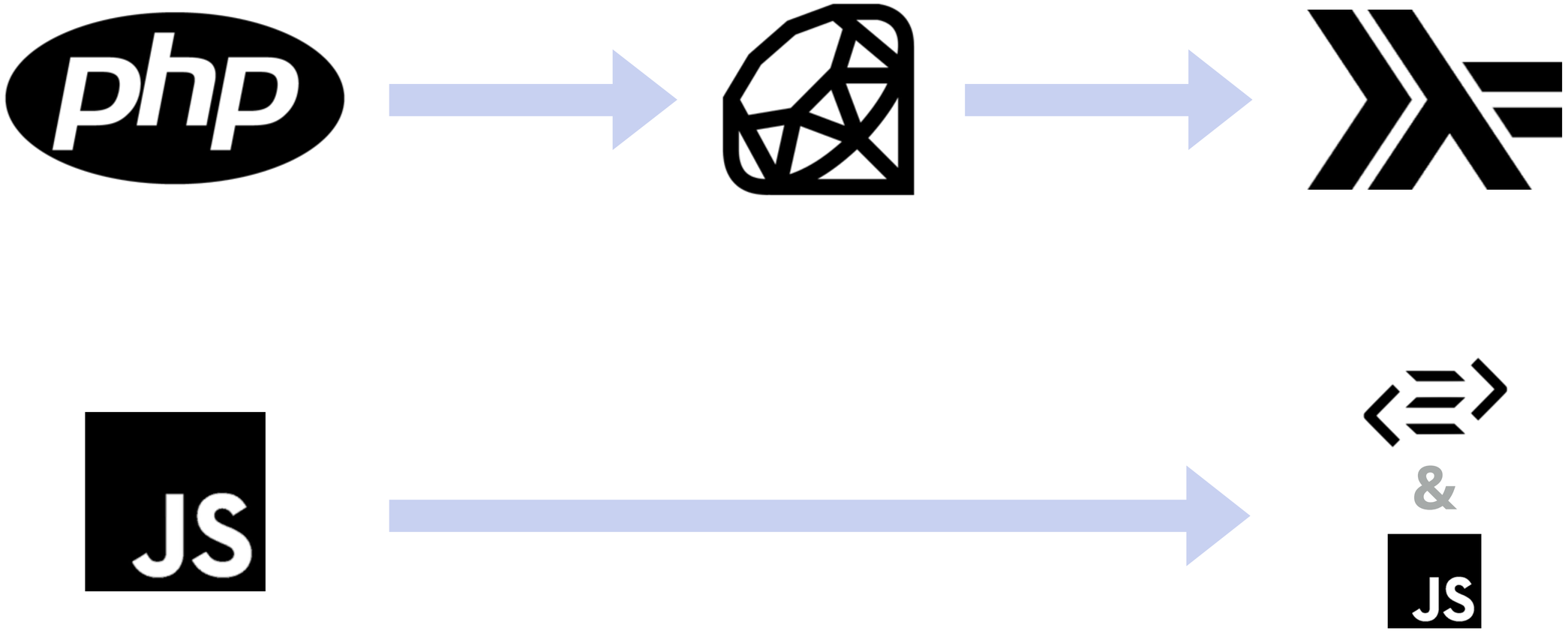

php


JS


php


JS


php
JS


php
JS
&
JS

"The competent programmer is fully aware of the strictly limited size of their own skull."

— Edsger W. Dijkstra

```
function unblockAll(batchSize, authDetails) {
  return makeTwitterClient(authDetails)
    .chain(client =>
      getBlocksList(client)
        .map(response => response.ids.map(
          id => unblockOne(client, id)
        )
    )
    .chain(reqs =>
      Future.parallel(batchSize, reqs)
    )
}
```

```
function unblockAll(batchSize, authDetails) {
  return makeTwitterClient(authDetails)
    .chain(client =>
      getBlocksList(client)
        .map(response => response.ids.map(
          id => unblockOne(client, id)
        )
    )
    .chain(reqs =>
      Future.parallel(batchSize, reqs)
    )
}
```

Referential Transparency

```
function unblockAll(batchSize, authDetails) {
  return makeTwitterClient(authDetails)
    .chain(client =>
      getBlocksList(client)
        .map(response => response.ids.map(
          id => unblockOne(client, id)
        )
    )
    .chain(reqs =>
      Future.parallel(batchSize, reqs)
    )
}
```

Immutability

```
function unblockAll(batchSize, authDetails) {
  return makeTwitterClient(authDetails)
    .chain(client =>
      getBlocksList(client)
        .map(response => response.ids.map(
          id => unblockOne(client, id)
        )
    )
    .chain(reqs =>
      Future.parallel(batchSize, reqs)
    )
}
```

Effects as Values

```
function unblockAll(batchSize, authDetails) {
  return makeTwitterClient(authDetails)
    .chain(client =>
      getBlocksList(client)
        .map(response => response.ids.map(
          id => unblockOne(client, id)
        )
    )
    .chain(reqs =>
      Future.parallel(batchSize, reqs)
    )
}
```

Types

Further Things to look into

Further Things to look into

Recursion

Further Things to look into

Recursion

Continuations

Further Things to look into

Recursion

Continuations

Pattern Matching (Decomposition)

Further Things to look into

Recursion

Continuations

Pattern Matching (Decomposition)

Parser-Combinator Libraries

Further Things to look into

Recursion

Continuations

Pattern Matching (Decomposition)

Parser-Combinator Libraries

Elm, Haskell and PureScript

Further Things to look into

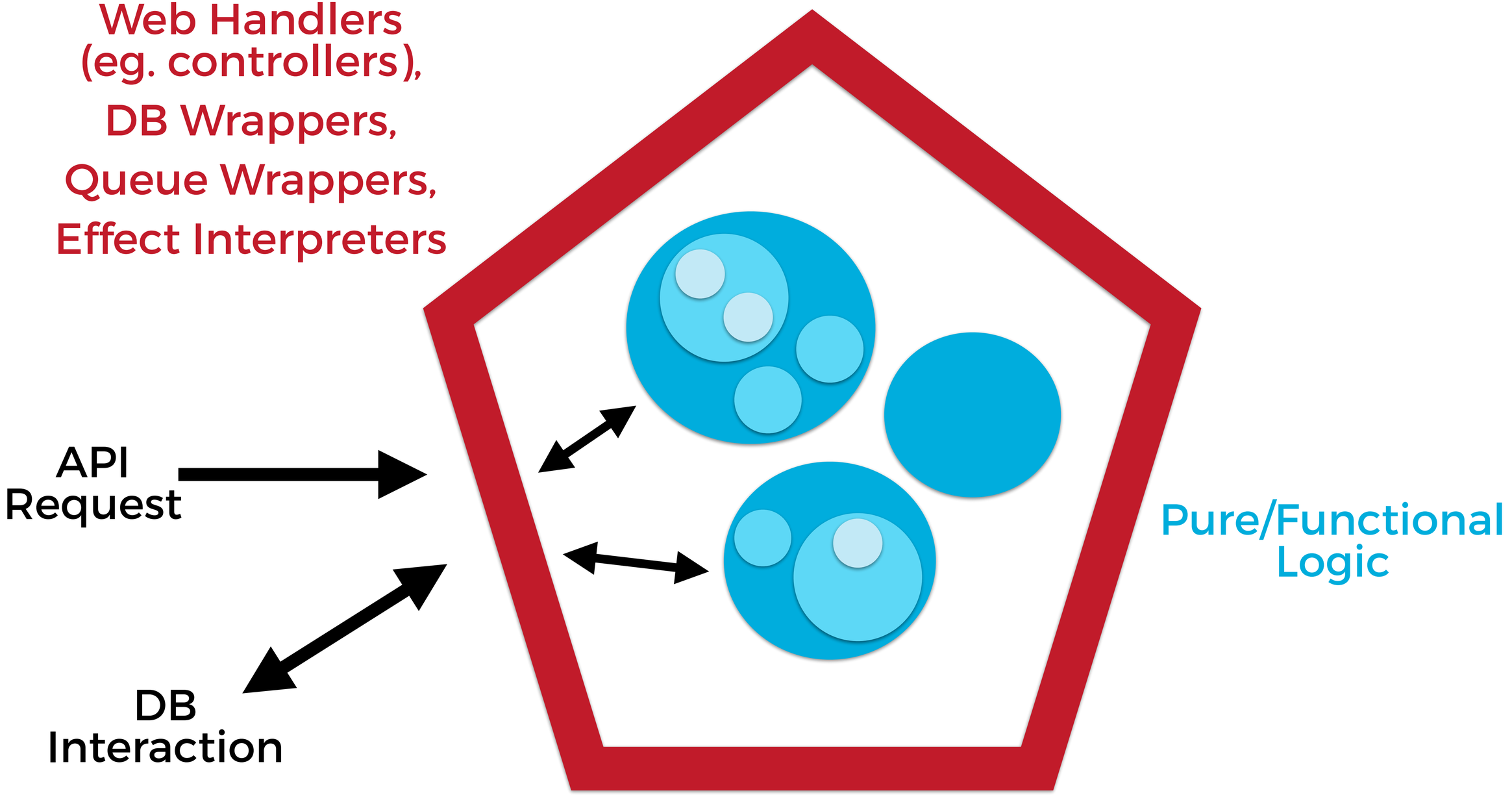
Recursion
Continuations
Pattern Matching (Decomposition)
Parser-Combinator Libraries
Elm, Haskell and PureScript
Lambda Calculus (a little bit)

“ ”

"Functional Core, Imperative [Effectful] Shell"

— Gary Bernhardt, "Boundaries"
https://www.youtube.com/watch?v=yTkzNHF6rMs

Web Handlers
(eg. controllers),
DB Wrappers,
Queue Wrappers,
Effect Interpreters



“ ”

"80% functional is better than 0%; you don't have to get to 99%."

— Gary Bernhardt, "Boundaries"
https://www.youtube.com/watch?v=yTkzNHF6rMs

What the Functional?

(What, Why, and How)

Rob Howard
@damncabbage
http://robhoward.id.au