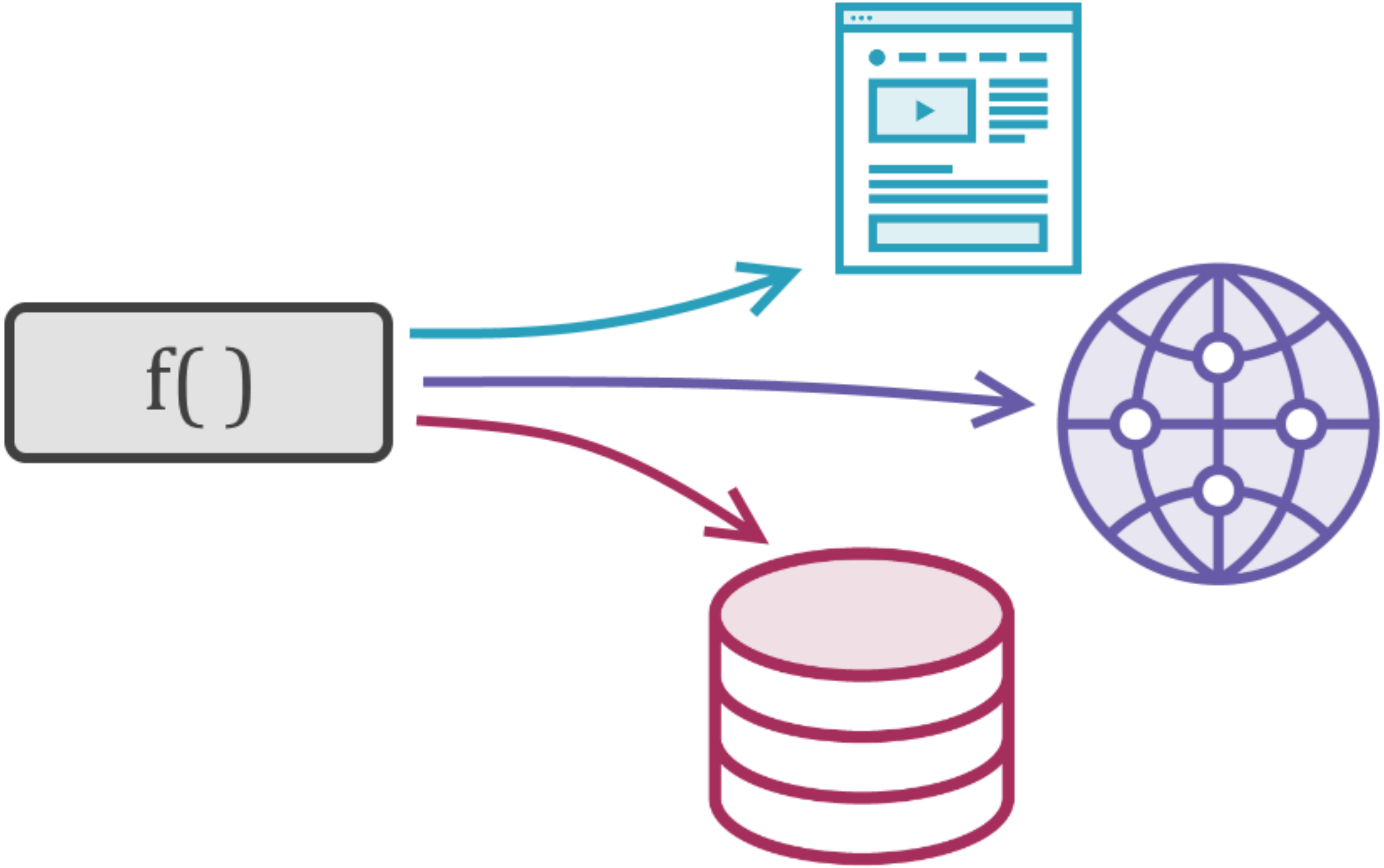
Introducing Pure Functions to Object Design

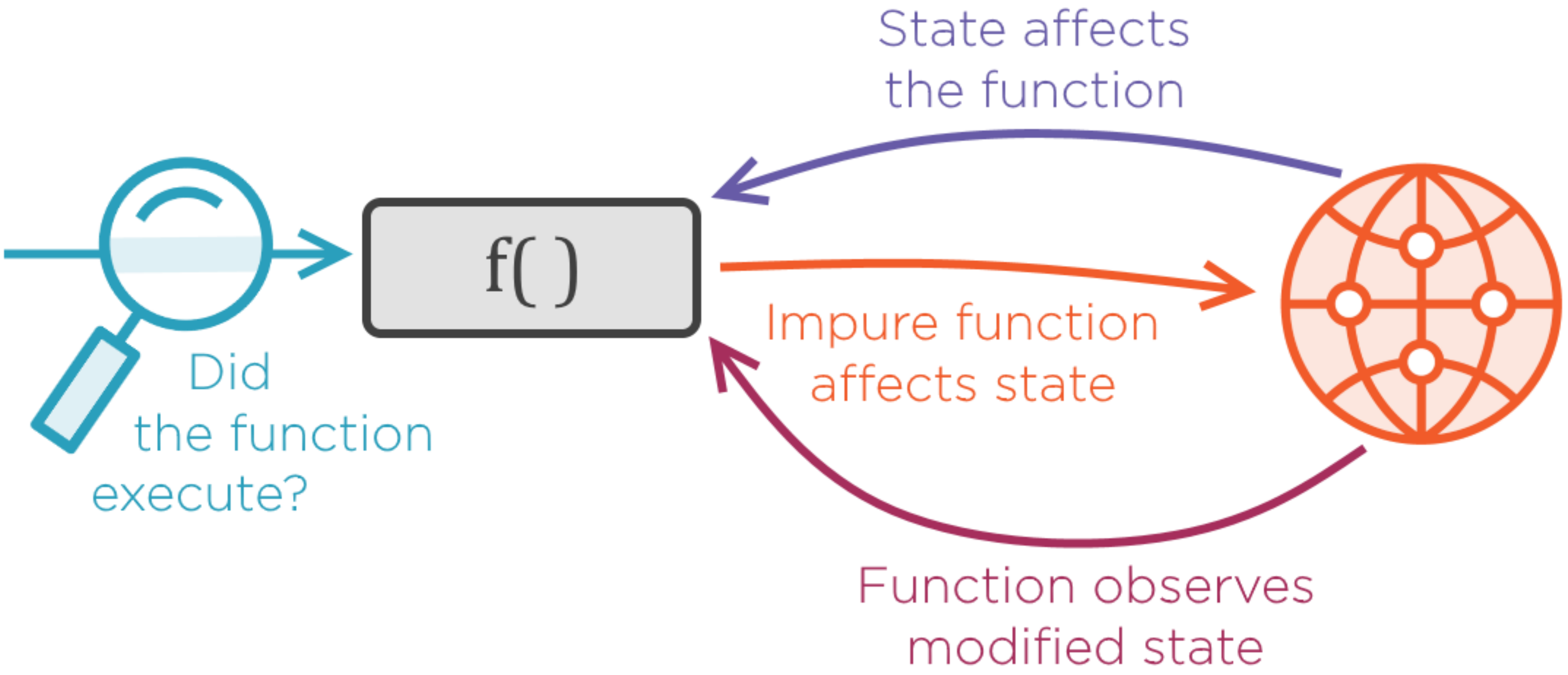
Zoran Horvat
CEO AT CODING HELMET

@zoranh75 http://csharpmentor.com

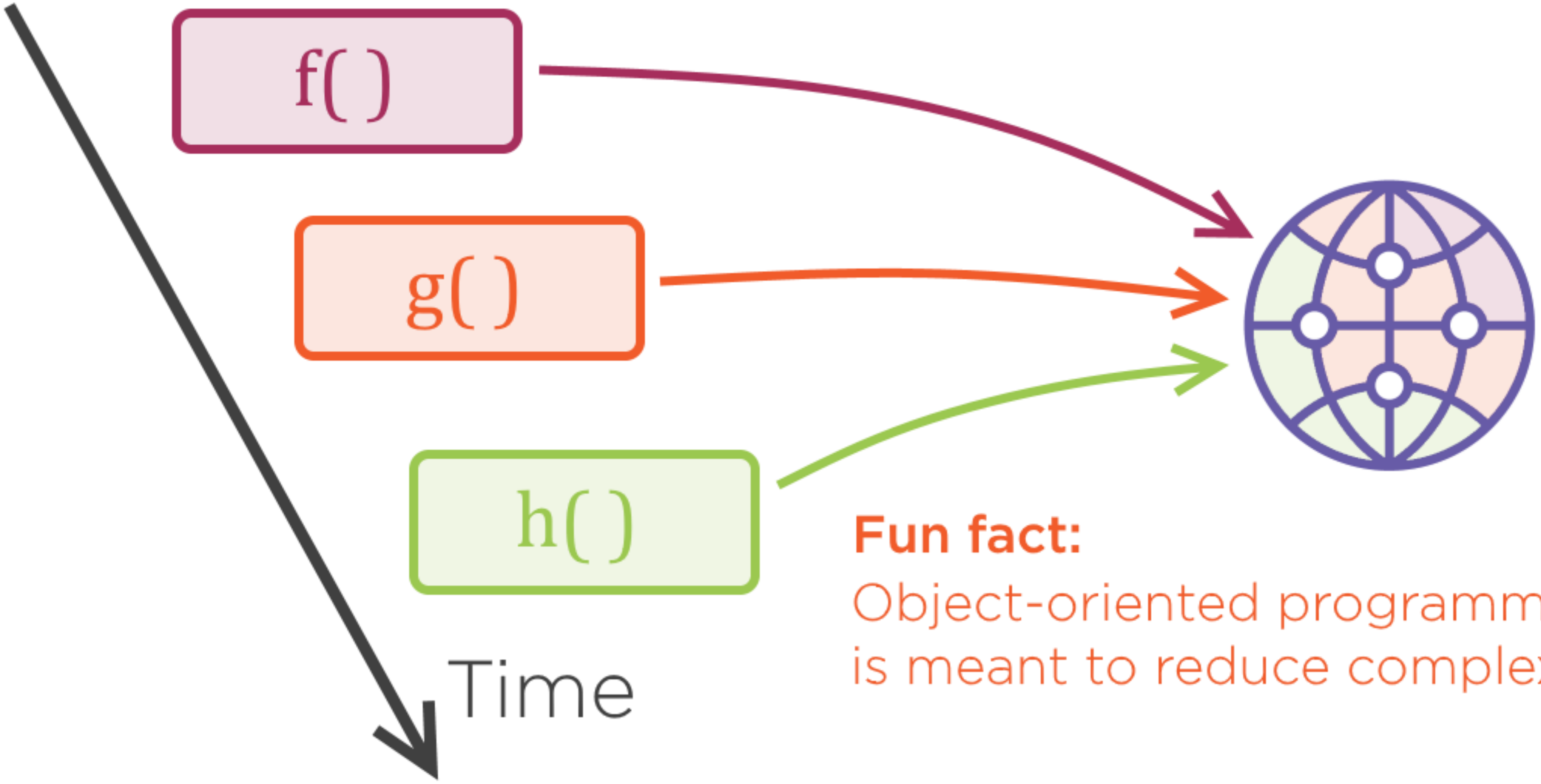
Understanding Side Effects



Understanding Side Effects


State affects the function
f( )
Did the function execute?
Impure function affects state
Function observes modified state

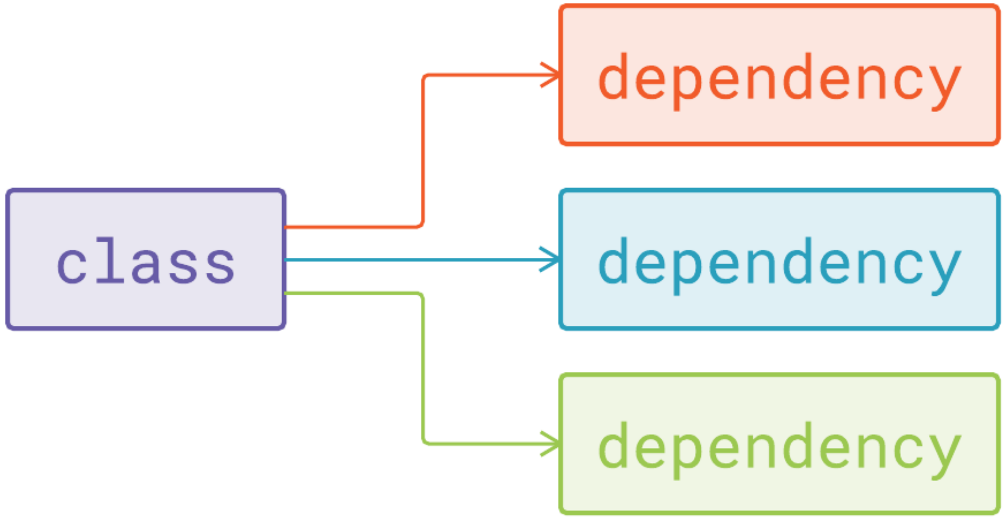
Understanding Side Effects



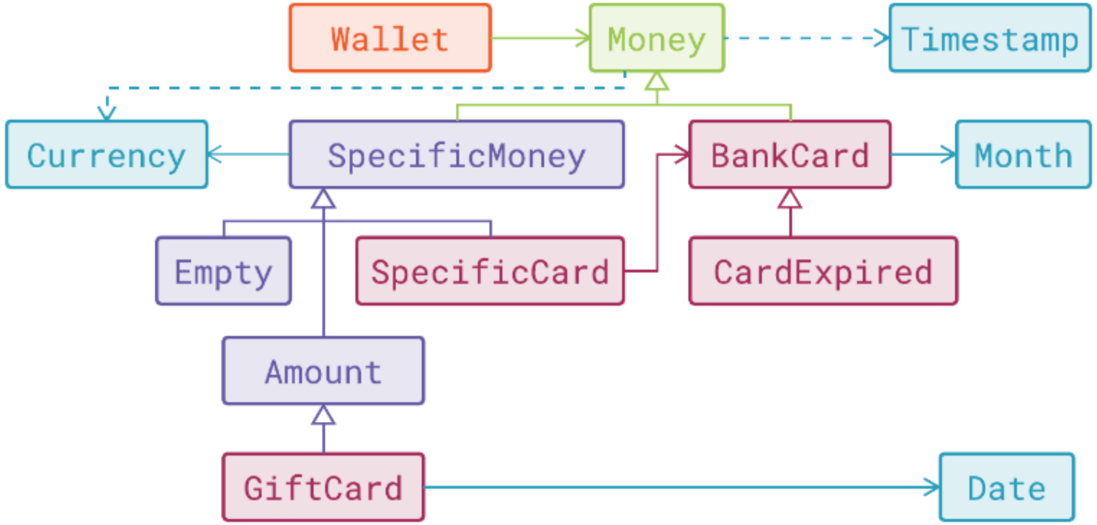
Fun fact:
Object-oriented programming
is meant to reduce complexity

Reducing Complexity

What Was Promised



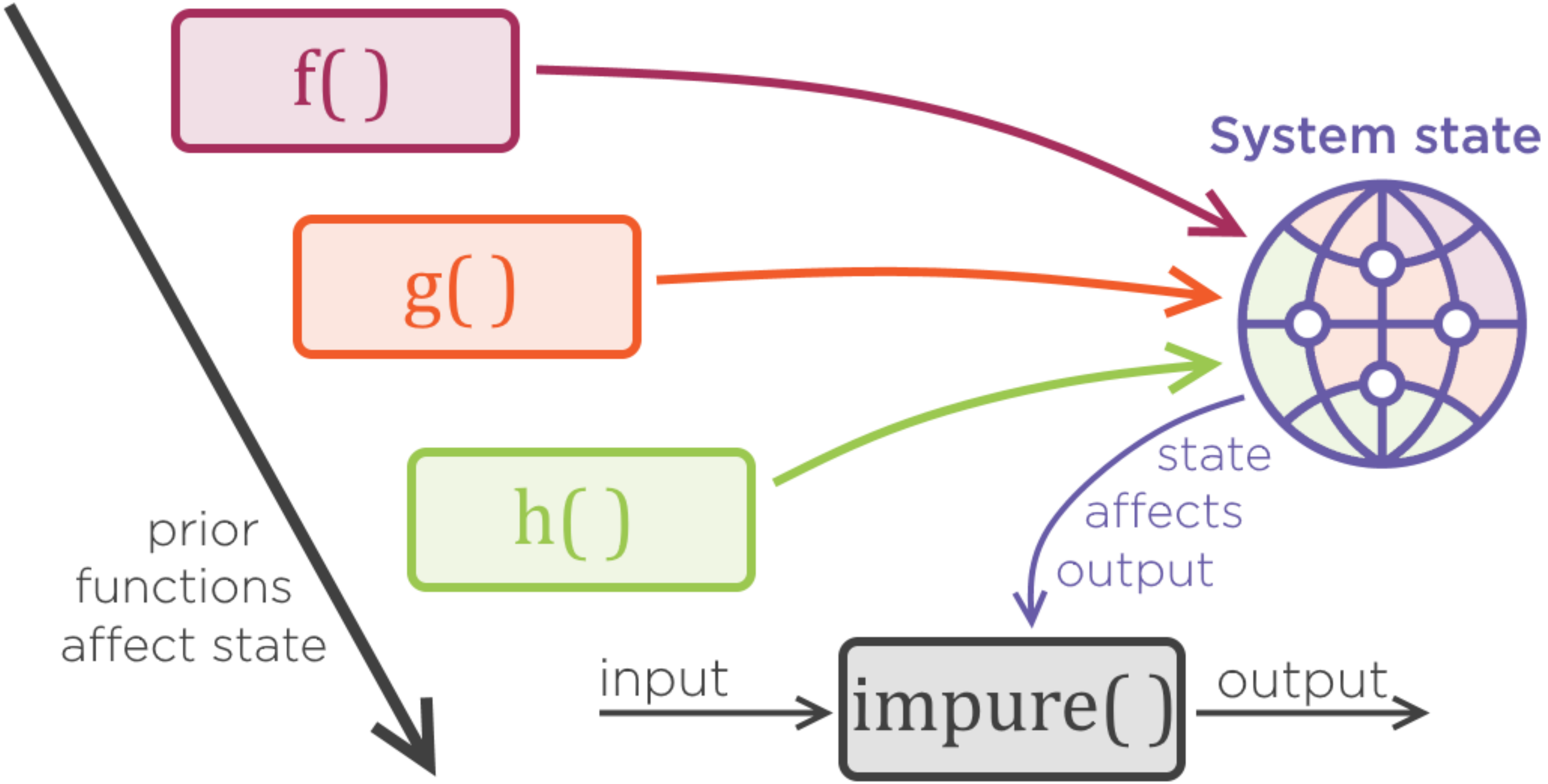
What We Have Got



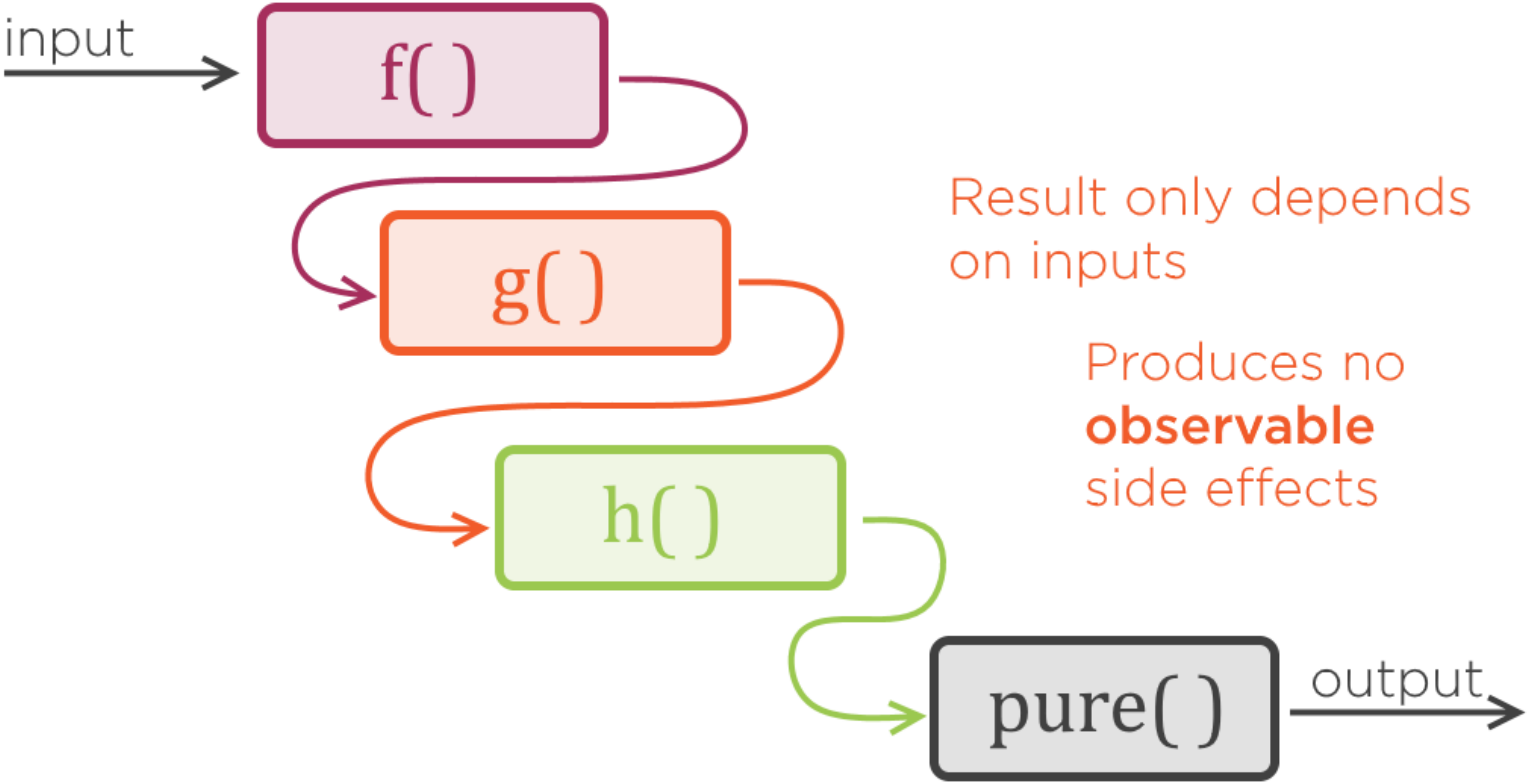
The fundamental interconnectedness of all things

Douglas Adams,
Dirk Gently's Holistic Detective Agency

Removing Side Effects



Removing Side Effects



Result only depends on inputs

Produces no **observable** side effects

Understanding Pure Functions

Will never modify its argument

Will never call a mutator on its argument

Will not have an `out` argument or in/out (`ref`) argument

Will never throw an exception

Will only tell its result through the return value

Then how do we return two things?

Understanding the **ValueTuple** Type

ValueTuple is a struct

Tuple is a class

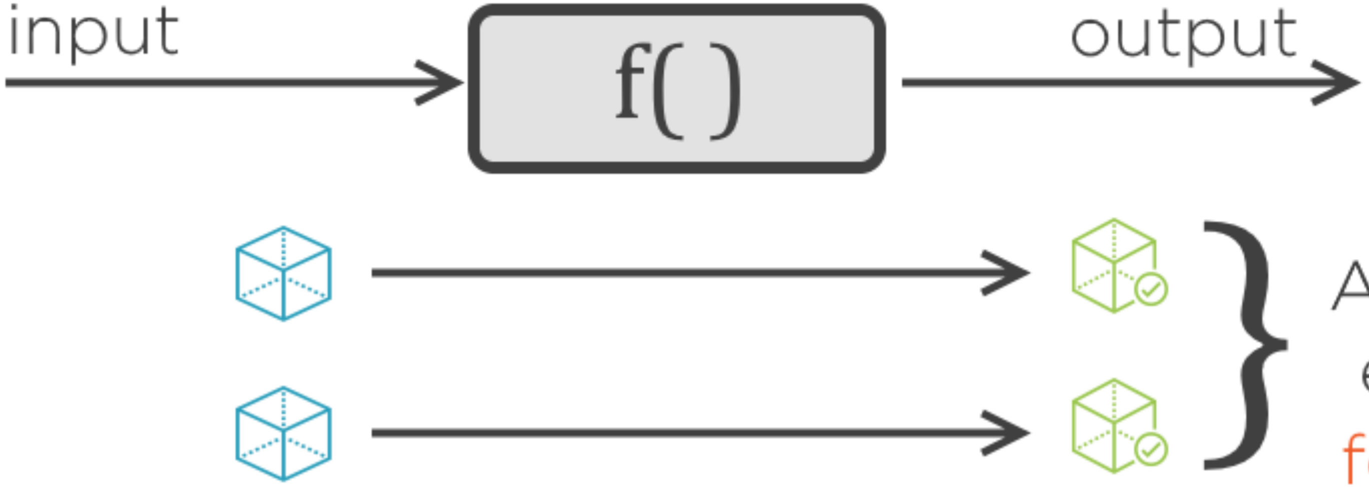
Components in **ValueTuple** are public mutable fields

Tuple exposes public read-only properties **Item1**, **Item2**, …

ValueTuple is not a good choice for public API

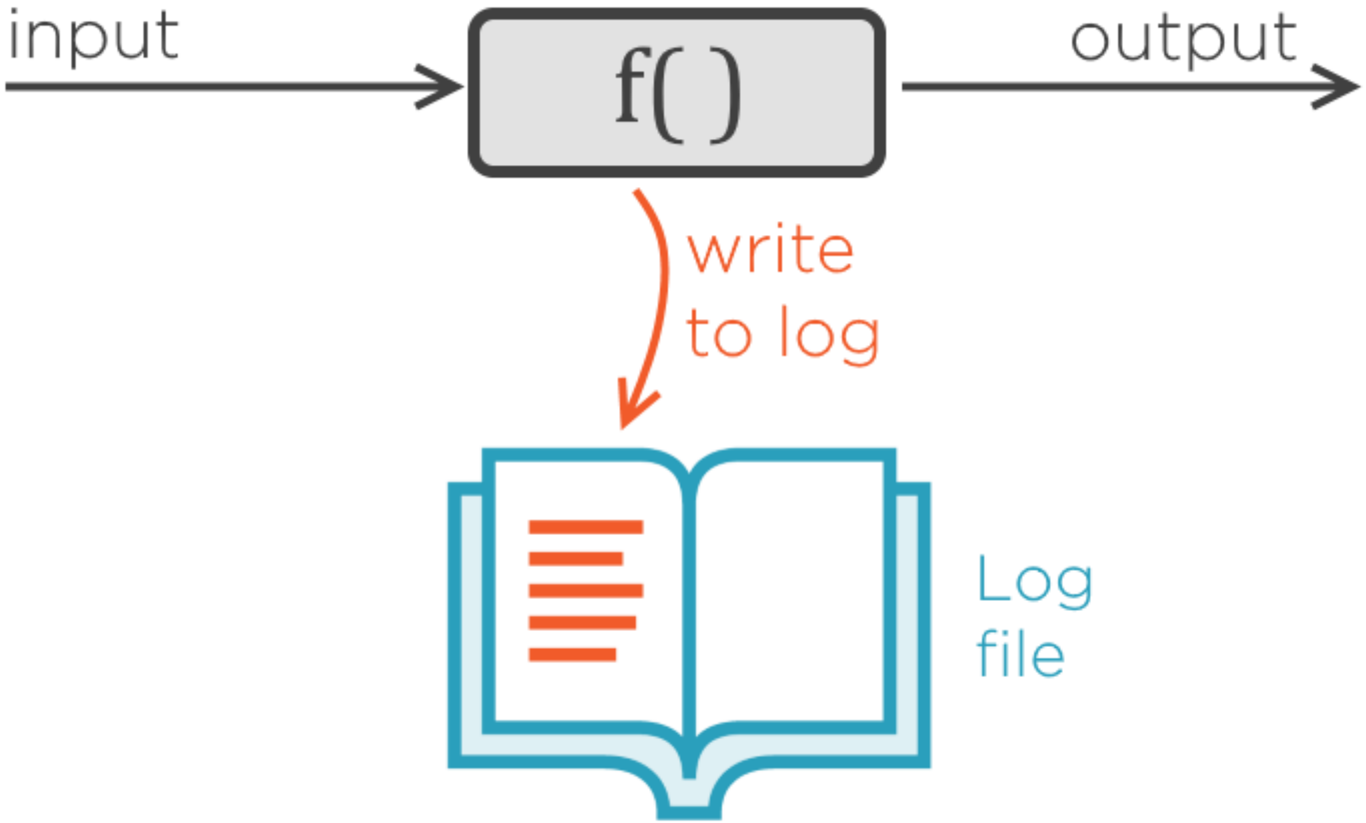
Don’t bind consumers to a struct with public fields!

Understanding Pure Functions

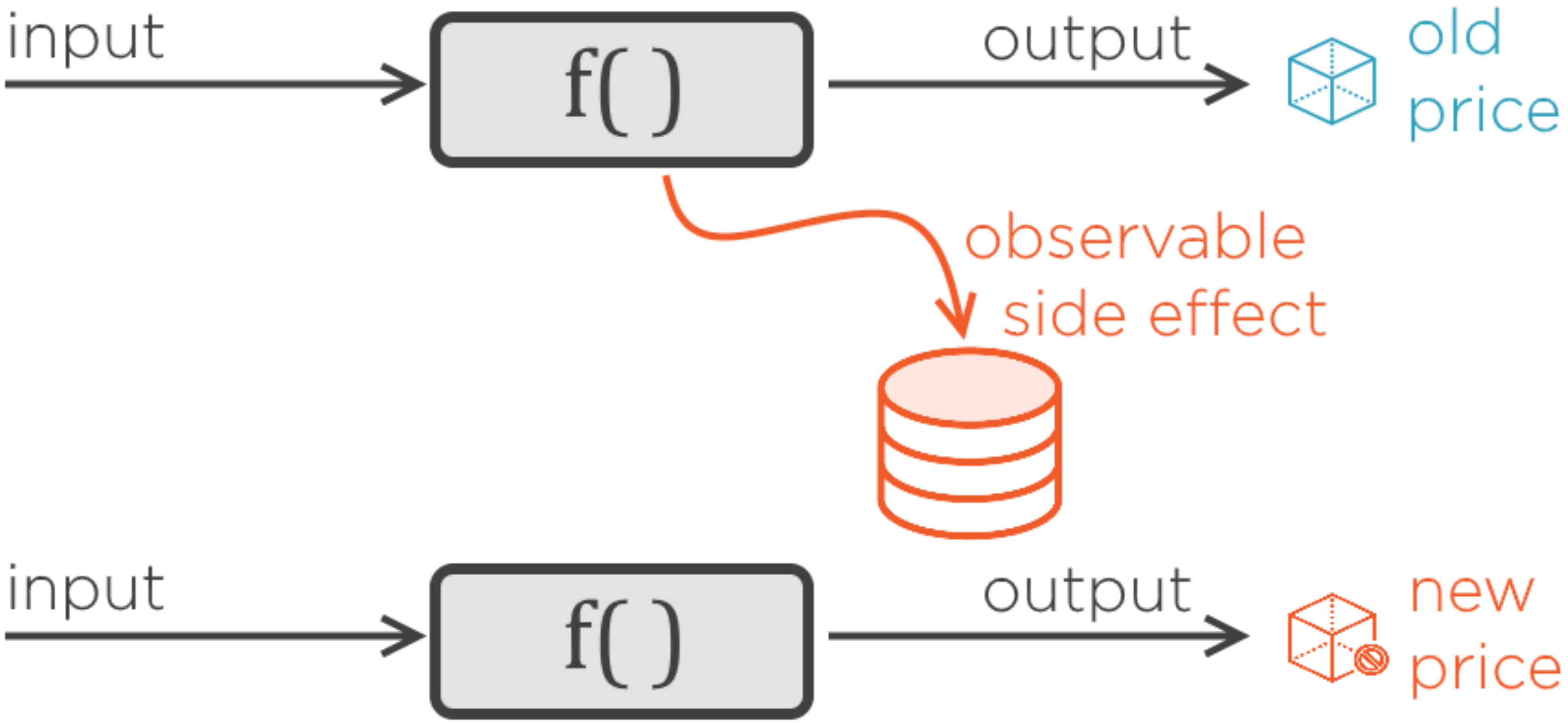


Always equal for the same input

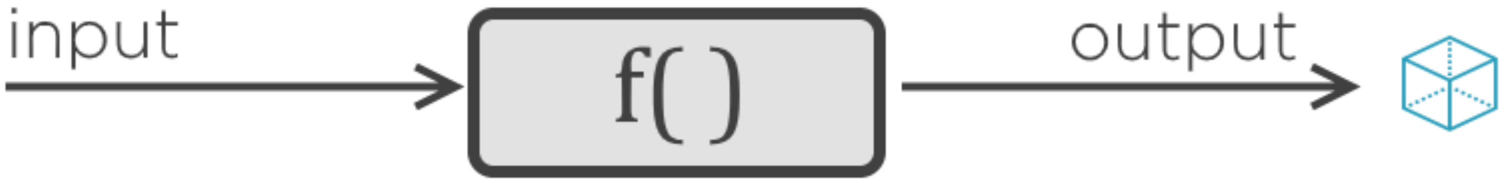
Understanding Pure Functions



Understanding Pure Functions



Understanding Pure Functions



an object returned

the same object

Referential Transparency

An expression is said to be referentially transparent if it can be replaced with its corresponding value without changing the program's behavior.

Wikipedia

Using Referential Transparency

result can be remembered and reused

```
class Amount
{
  ...
  public (Amount taken, Money remaining) Take(decimal amount)
  {
    decimal taken = Math.Min(this.Value, amount);
    decimal remaining = this.Value - taken;

    return
    (
        new Amount(base.Currency, taken),
        new Amount(base.Currency, remaining)
    );
  }
  ...
}
```

pure function

Using Referential Transparency

```
class Amount
{
   ...
   public (Amount taken, Money remaining) Take(decimal amount) { ... }
   ...
}
```

result can be remembered and reused

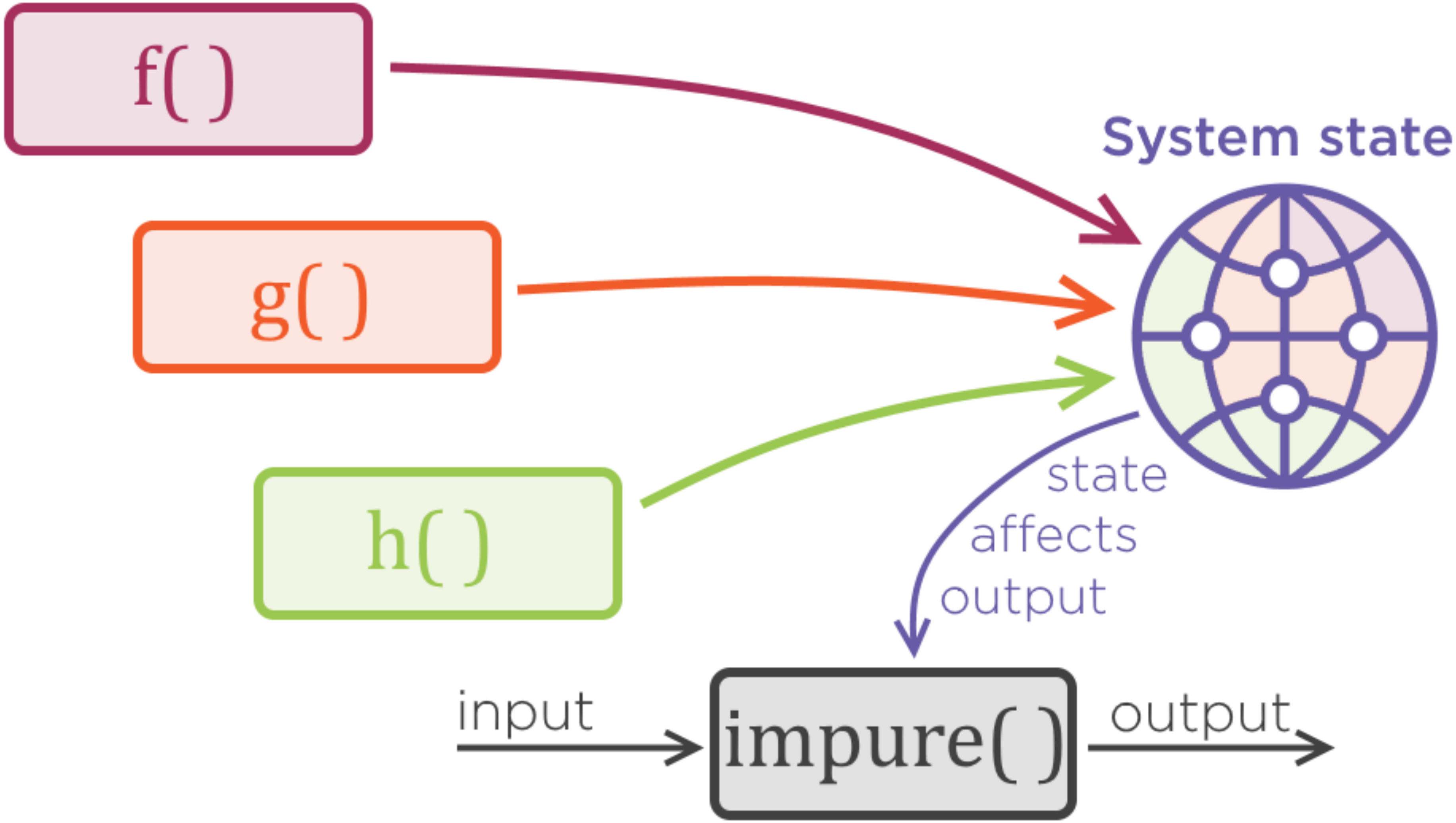
```
    tuple1 = amount.Take(fiveDollars);
    tuple2 = amount.Take(fiveDollars);
```

← *bad idea*

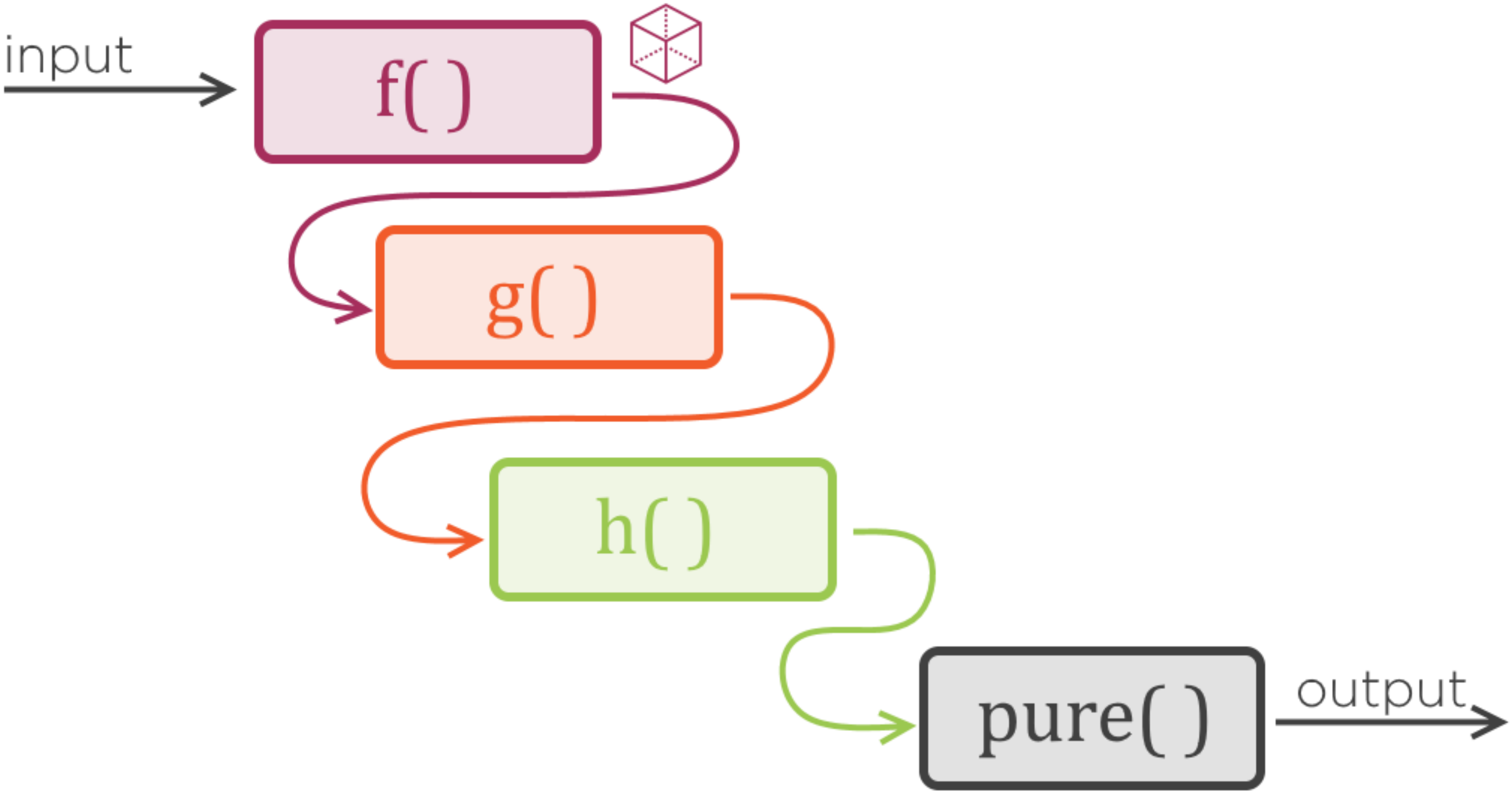
```
        tuple1 = amount.Take(fiveDollars);
        tuple2 = tuple1;
```

← *reuse previous result*

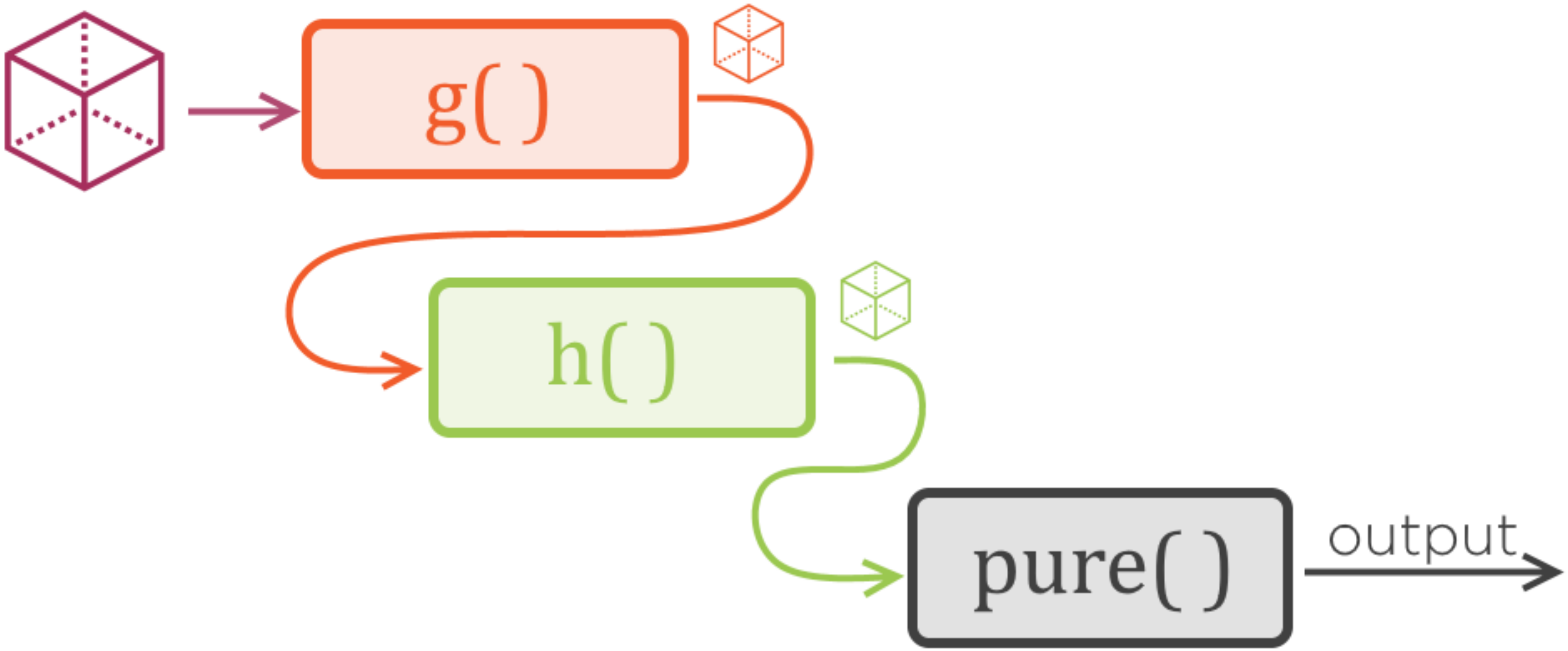
Using Referential Transparency



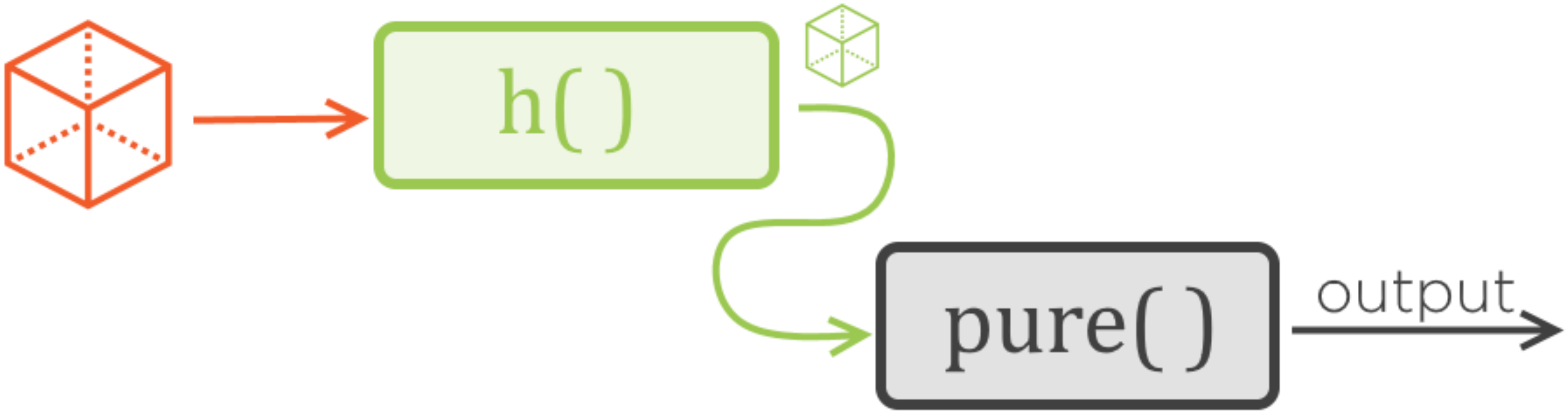
Using Referential Transparency



Using Referential Transparency

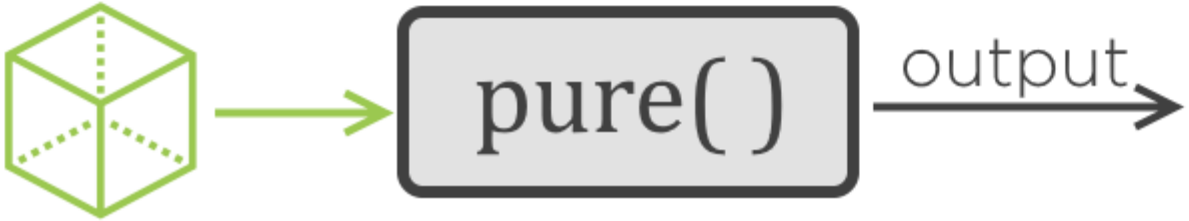


Using Referential Transparency

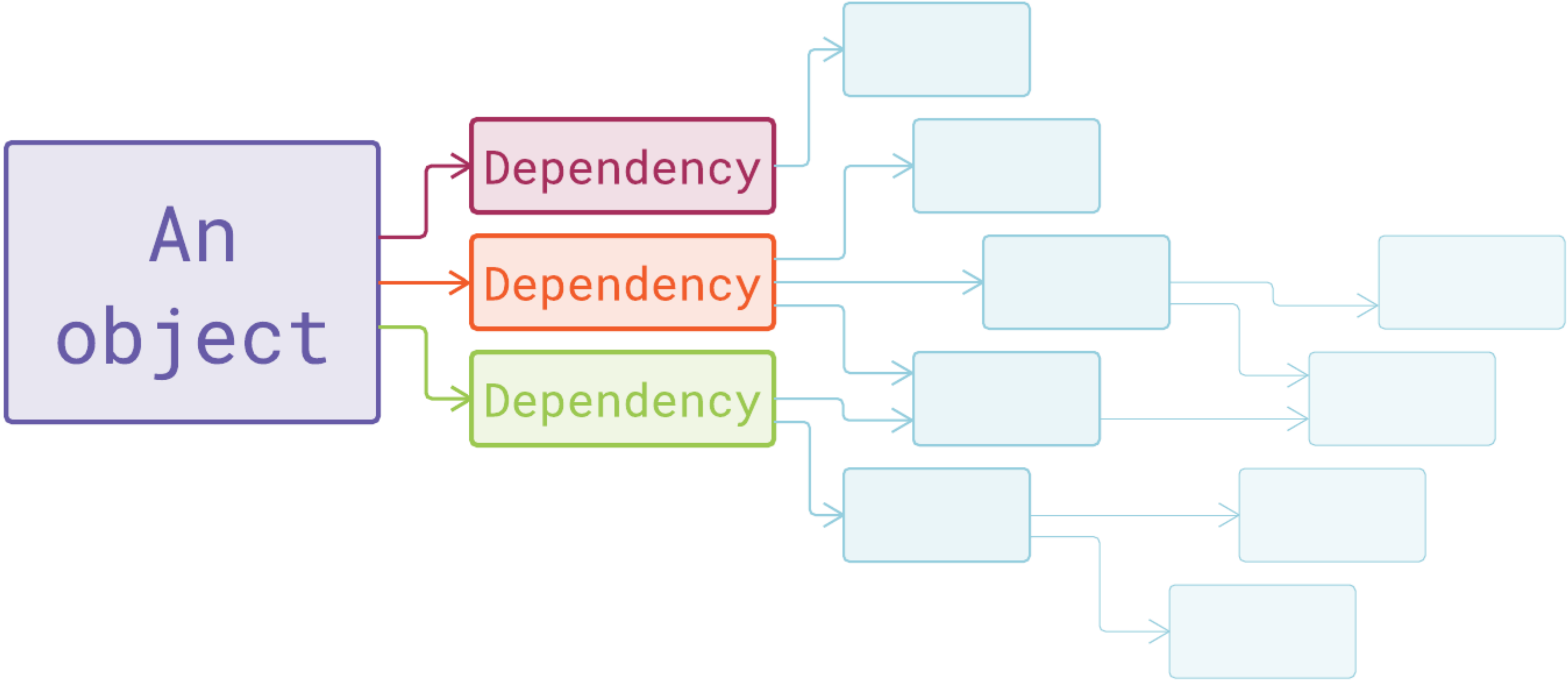


Using Referential Transparency

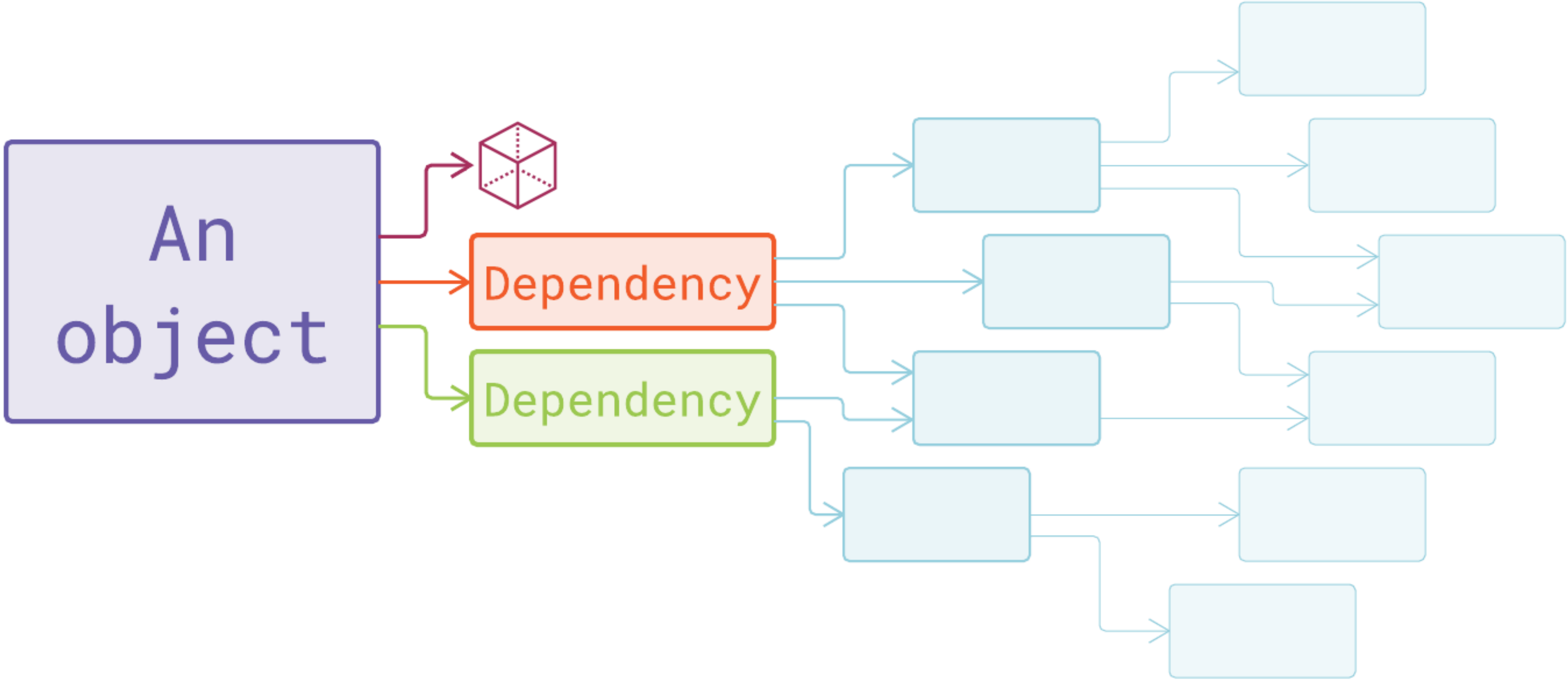
All complexity has just vanished



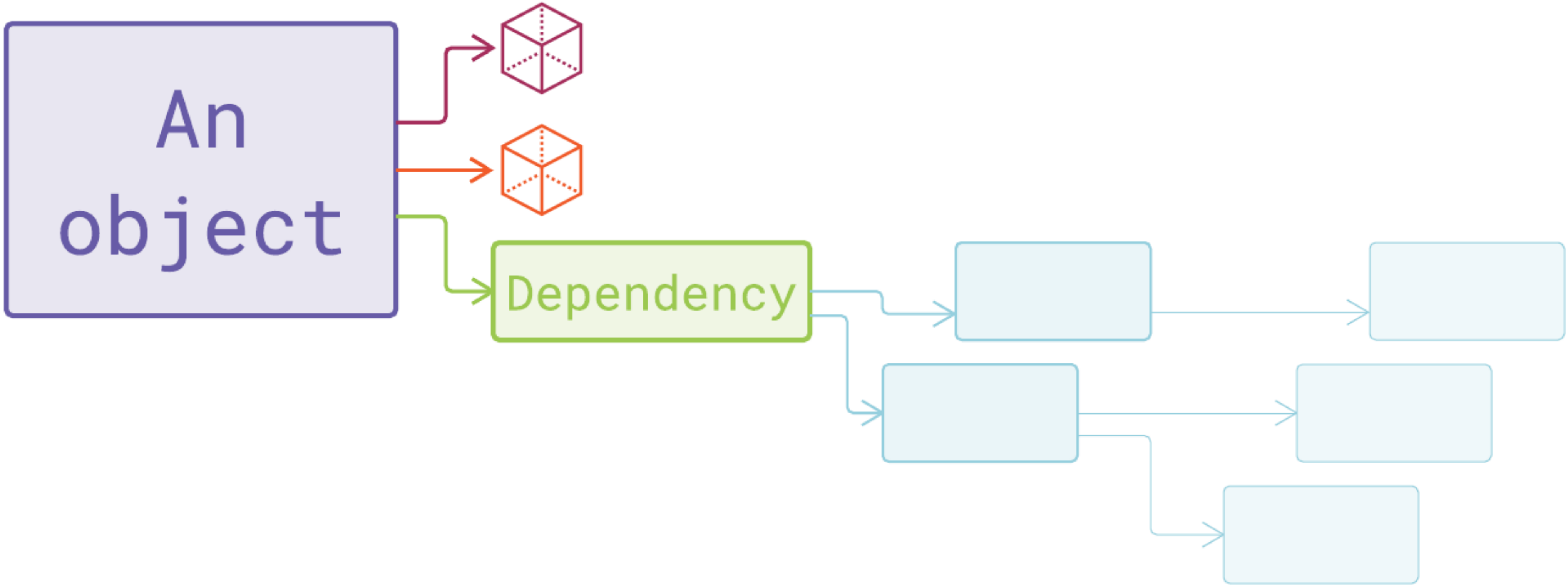
Purity in Deep Object Models



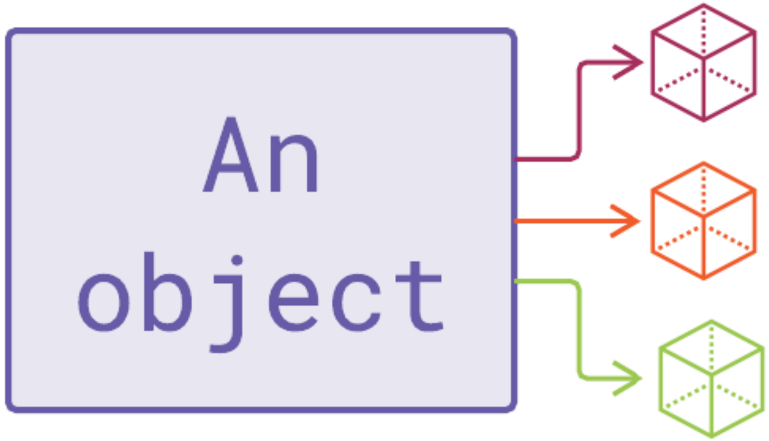
Purity in Deep Object Models



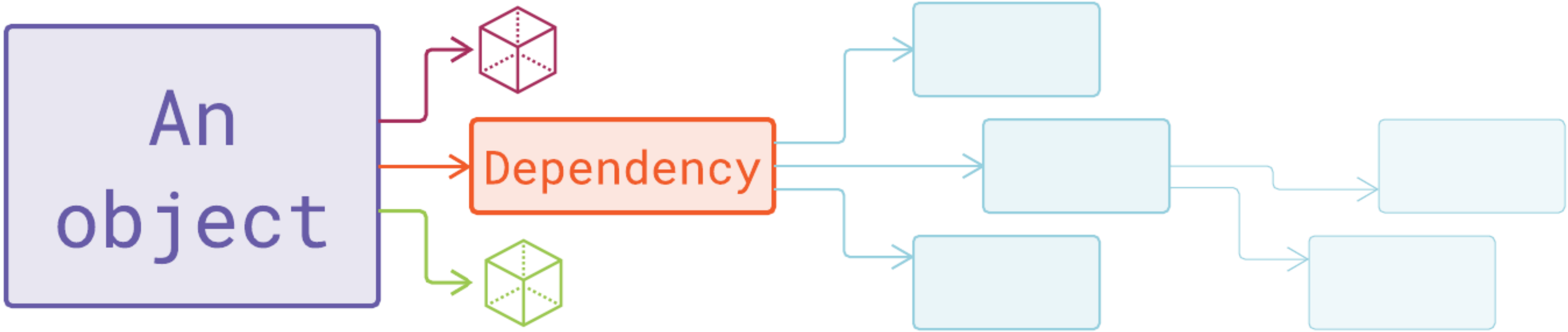
Purity in Deep Object Models



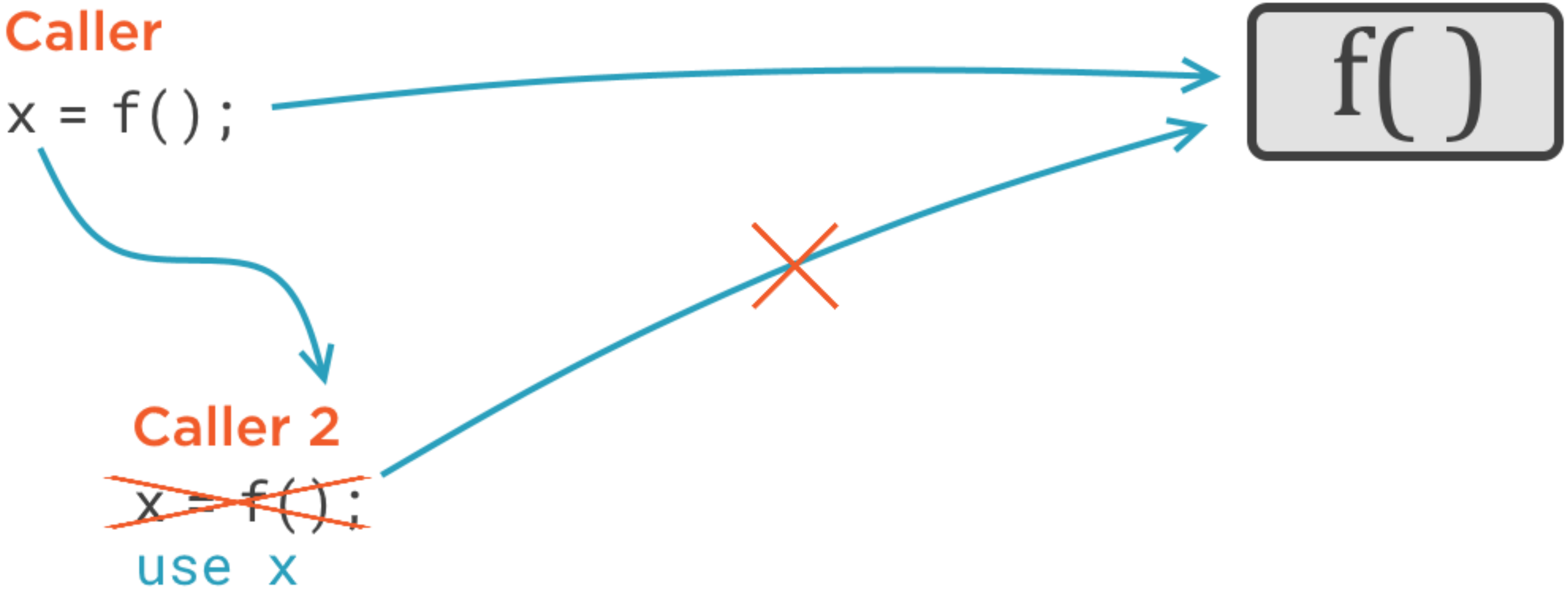
Purity in Deep Object Models



Purity in Deep Object Models



Inventing Memoization



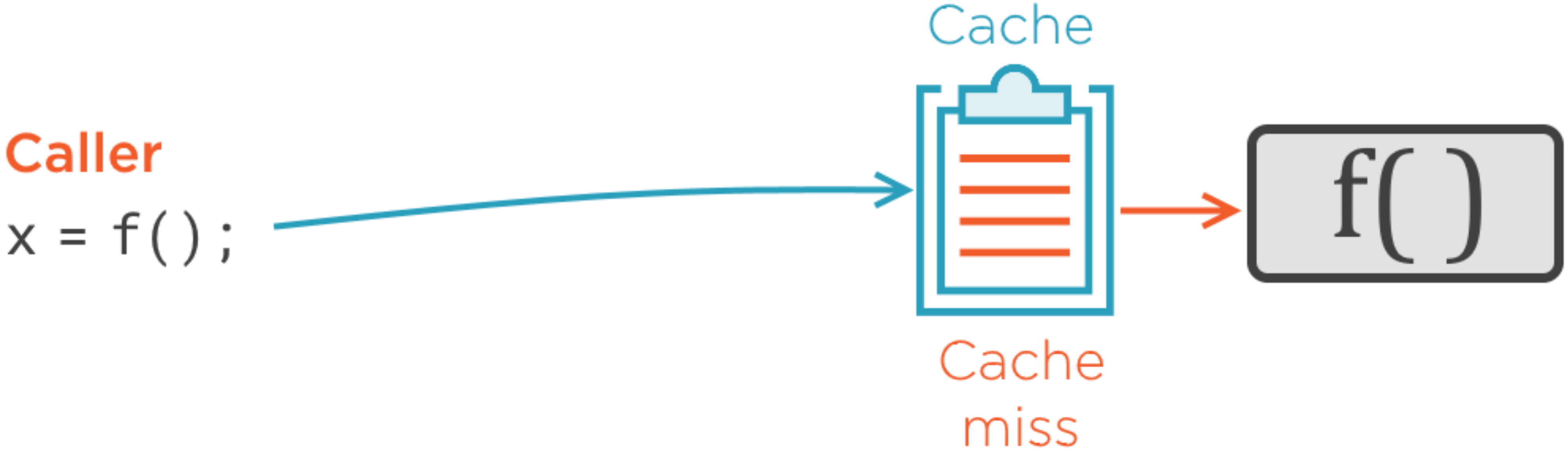
Caller

```
x = f();
```

Caller 2

```
~~x = f();~~
use x
```

Inventing Memoization



Caller

```
x = f();
```

Cache

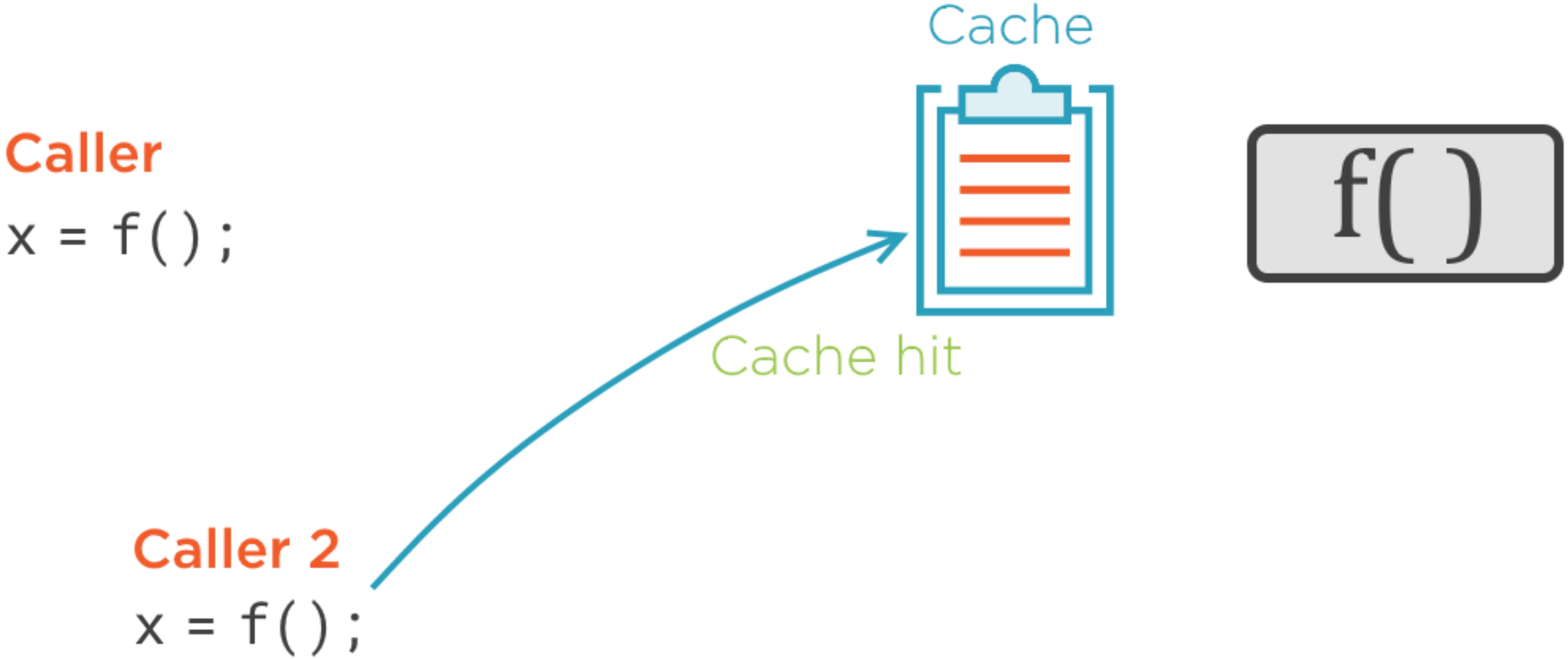
Cache miss

f( )

Caller 2

```
x = f();
```

Inventing Memoization



Caller

```
x = f();
```

Caller 2

```
x = f();
```

Cache

Cache hit

Inventing Memoization

Caller

```
x = f();
```

Cache

f( )

Caller 2

```
x = f();
```

Memoization

Cache results to avoid repeated function evaluations

Demo

Memoization Example

Fibonacci sequence

$F_1=1,\ F_2=1,\ F_n=F_{n-1}+F_{n-2}\ (n > 2)$

1, 1, 2, 3, 5, 8, 13...

Zero-based Fibonacci sequence

$F_0=0,\ F_1=1,\ F_n=F_{n-1}+F_{n-2}\ (n > 1)$

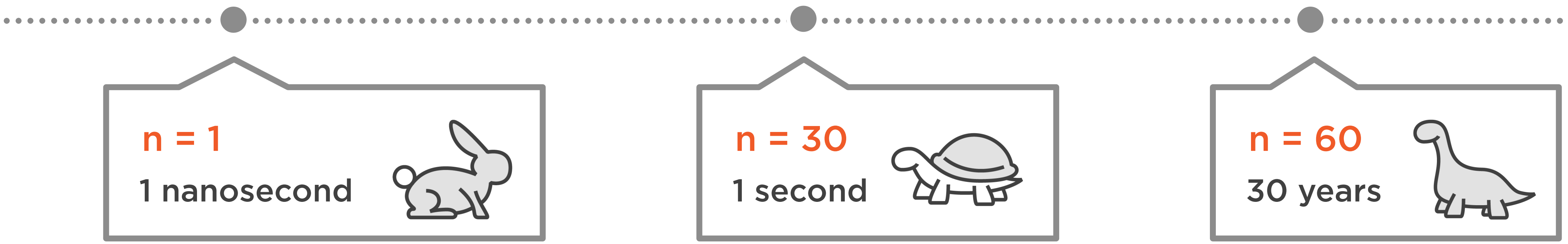
0, 1, 1, 2, 3, 5, 8, 13...

Fibonacci Sequence Performance

$$F_n = F_{n-1} + F_{n-2}$$

$$time_n = time_{n-1} + time_{n-2}$$

$$2 \cdot time_{n-2} < time_n < 2 \cdot time_{n-1}$$



n = 1

1 nanosecond

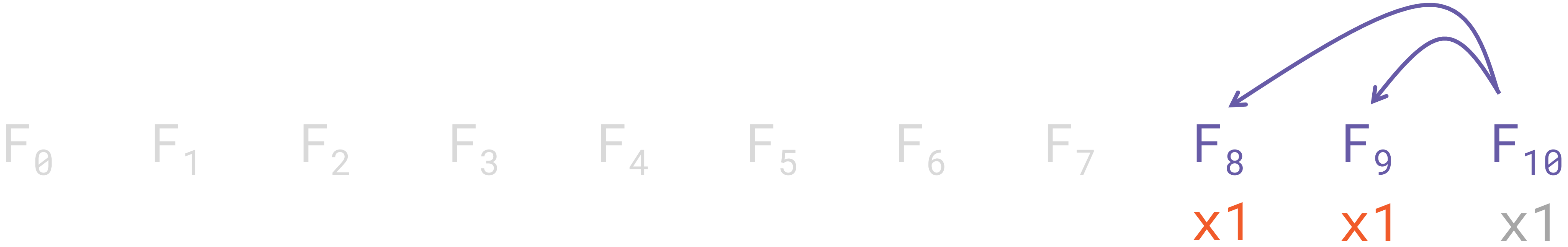
n = 30

1 second

n = 60

30 years

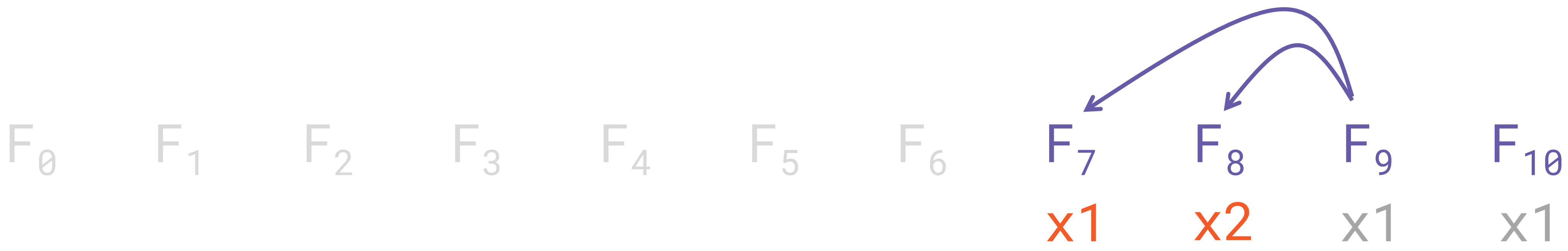
Fibonacci Sequence Performance



F_0 F_1 F_2 F_3 F_4 F_5 F_6 F_7 F_8 F_9 F_{10}

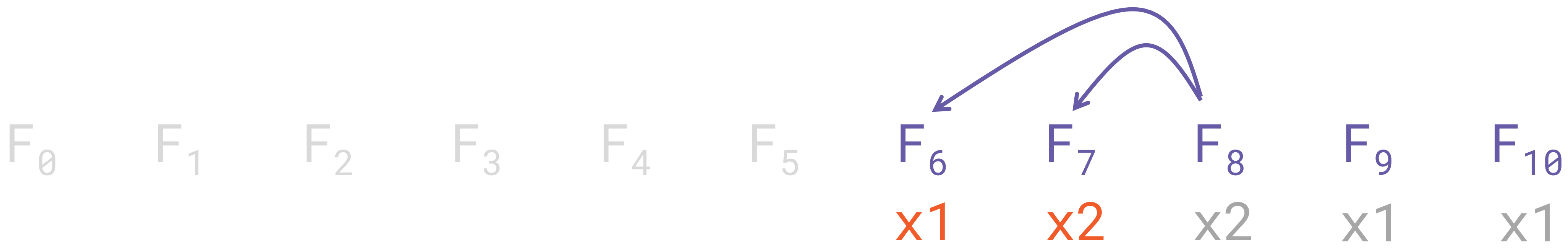
x1 x1 x1

Fibonacci Sequence Performance



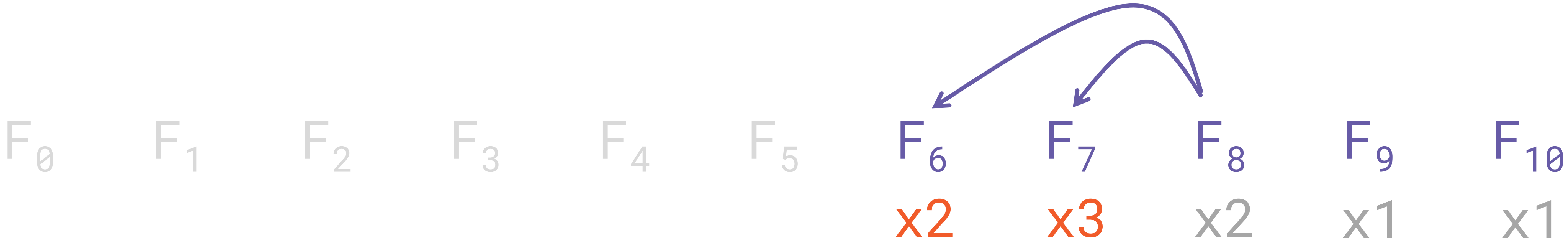
F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
							x1	x2	x1	x1

Fibonacci Sequence Performance



F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
						x1	x2	x2	x1	x1

Fibonacci Sequence Performance



F_0 F_1 F_2 F_3 F_4 F_5 F_6 F_7 F_8 F_9 F_{10}

x2 x3 x2 x1 x1

Fibonacci Sequence Performance

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
x34	x55	x34	x21	x13	x8	x5	x3	x2	x1	x1

F_{20} 6 thousand calls
F_{30} 800 thousand calls
F_{40} 100 million calls
F_{50} 12.5 billion calls
F_{60} 1.5 trillion calls

Should be:

F_{20} 1 call
F_{30} 1 call
F_{40} 1 call
F_{50} 1 call
F_{60} 1 call

Summary

Programmatic functions

- View them as values they produce

Pure function

- Has no observable side effects
- Return value only depends on arguments
- Referentially transparent

Memoization

- Cache results of a pure function
- Useful when calling the same pure function many times with same input

Next module:
Memoization with Pure Functions