

HOW TO BE A FUNCTIONAL PROGRAMMER

without being a jerk about it

NATHAN DOTZ • @NATHANDOTZ • CODEMASH 2014

MONAD

FUNCTION

ZYGOHISTOMORPHIC PREPROMORPHISMS



You're a jerk.

Nathan Dotz (sleepynate)

I F#
ON THE
FIRST
DATE

**IMPERATIVE
PROGRAMMING
OBVIOUSLY
SUCKS**

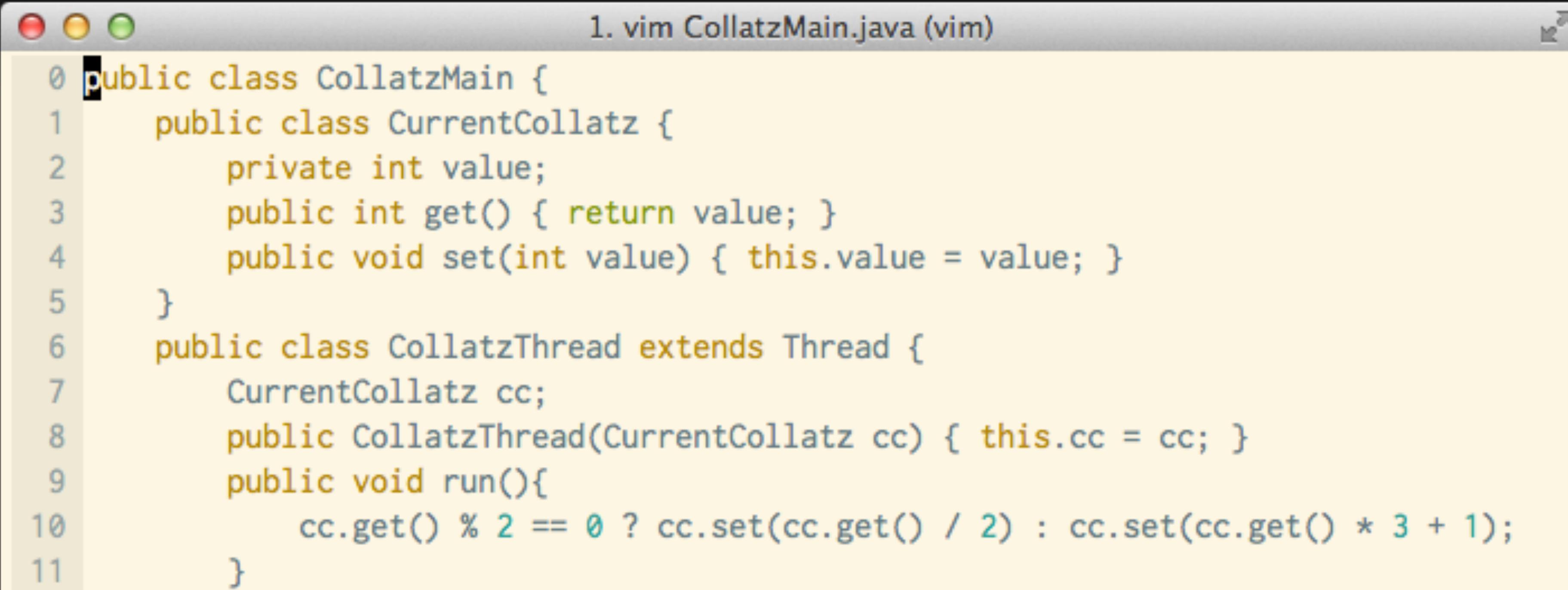
Damn, this t-shirt is ugly



WHY DOES IMPERATIVE SUCK?

- Mutability
- Break, continue, goto. Where's the control?
- Assignment and dereferencing

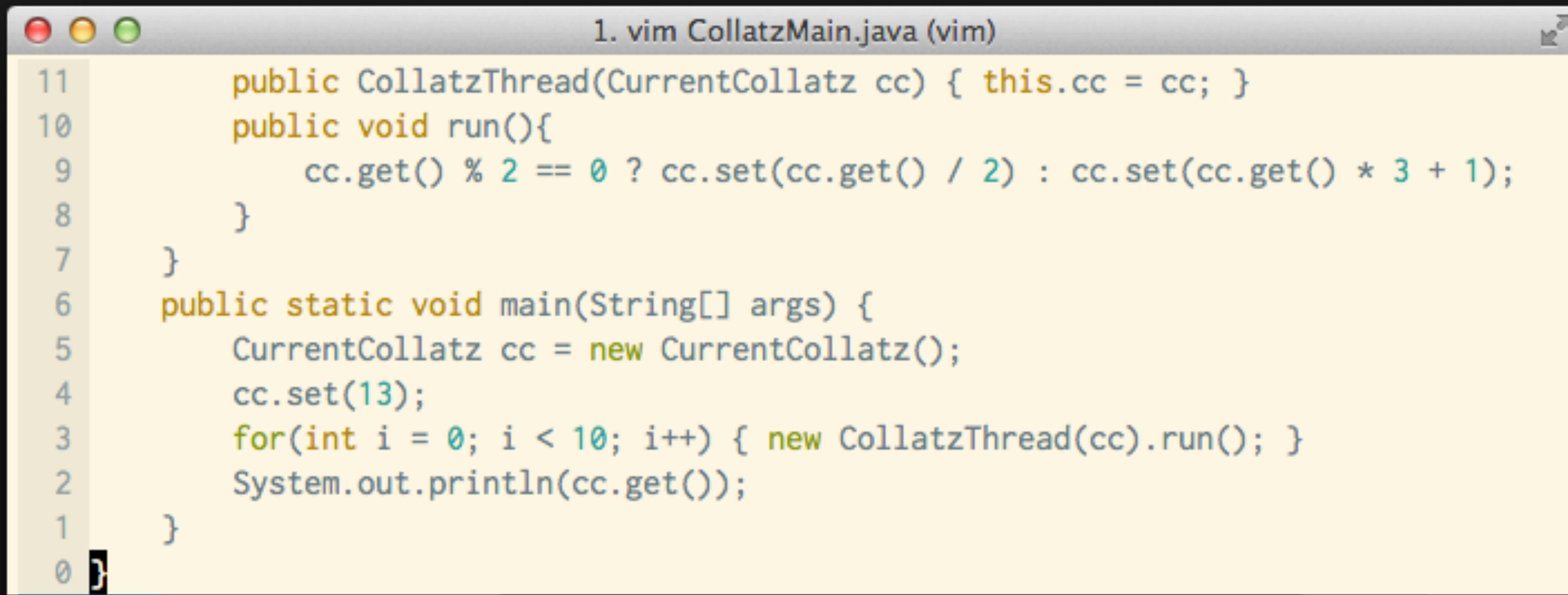
IMPERATIVE SUCKS ‘CAUSE MUTABILITY?



A screenshot of a Mac OS X terminal window titled "1. vim CollatzMain.java (vim)". The window shows Java code for a Collatz sequence calculator. The code defines a `CurrentCollatz` class with a private `value` field and `get` and `set` methods. It also defines a `CollatzThread` class that extends `Thread` and overrides the `run` method to modify the `value` of a `CurrentCollatz` object. The code is numbered from 0 to 11.

```
0 public class CollatzMain {
1     public class CurrentCollatz {
2         private int value;
3         public int get() { return value; }
4         public void set(int value) { this.value = value; }
5     }
6     public class CollatzThread extends Thread {
7         CurrentCollatz cc;
8         public CollatzThread(CurrentCollatz cc) { this.cc = cc; }
9         public void run(){
10             cc.get() % 2 == 0 ? cc.set(cc.get() / 2) : cc.set(cc.get() * 3 + 1);
11         }
12     }
13 }
```

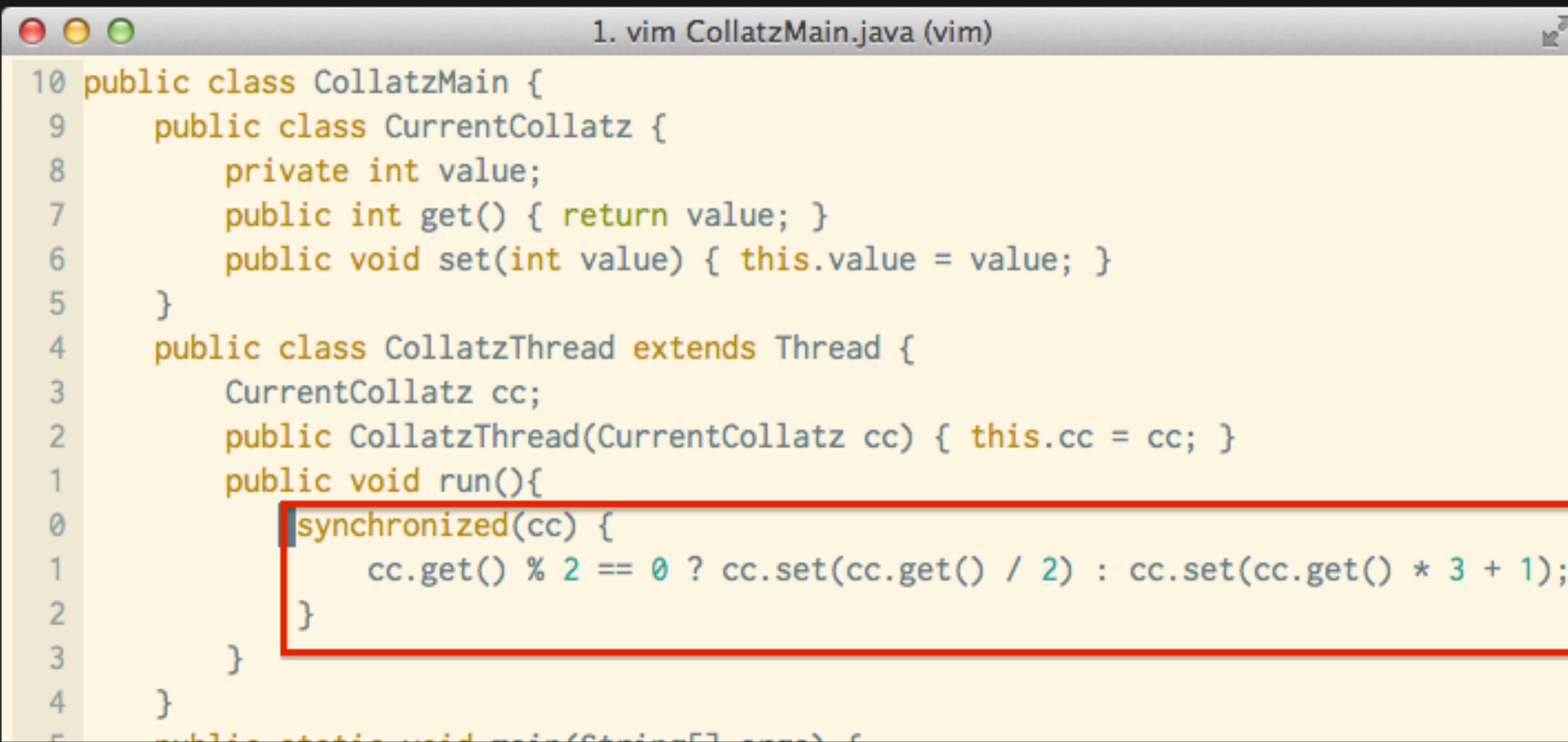
IMPERATIVE SUCKS ‘CAUSE MUTABILITY?



The image shows a Mac OS X terminal window with a vim editor session. The title bar reads "1. vim CollatzMain.java (vim)". The code in the editor is:

```
11     public CollatzThread(CurrentCollatz cc) { this.cc = cc; }
10     public void run(){
9         cc.get() % 2 == 0 ? cc.set(cc.get() / 2) : cc.set(cc.get() * 3 + 1);
8     }
7 }
6     public static void main(String[] args) {
5         CurrentCollatz cc = new CurrentCollatz();
4         cc.set(13);
3         for(int i = 0; i < 10; i++) { new CollatzThread(cc).run(); }
2         System.out.println(cc.get());
1     }
0 }
```

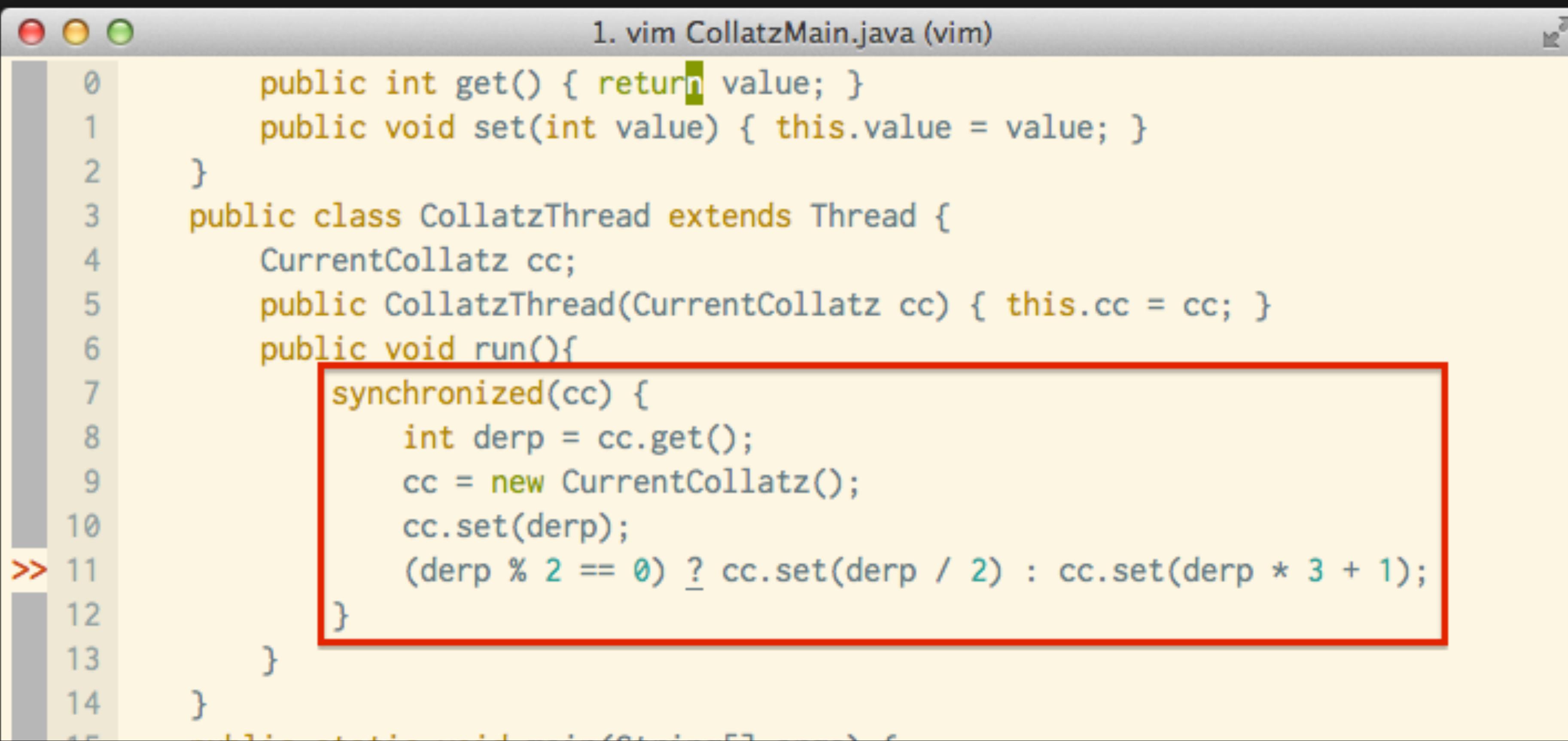
IMPERATIVE SUCKS ‘CAUSE MUTABILITY?



```
1. vim CollatzMain.java (vim)
10 public class CollatzMain {
9     public class CurrentCollatz {
8         private int value;
7         public int get() { return value; }
6         public void set(int value) { this.value = value; }
5     }
4     public class CollatzThread extends Thread {
3         CurrentCollatz cc;
2         public CollatzThread(CurrentCollatz cc) { this.cc = cc; }
1         public void run(){
0             synchronized(cc) {
1                 cc.get() % 2 == 0 ? cc.set(cc.get() / 2) : cc.set(cc.get() * 3 + 1);
2             }
3         }
4     }
5     public static void main(String[] args) {

```

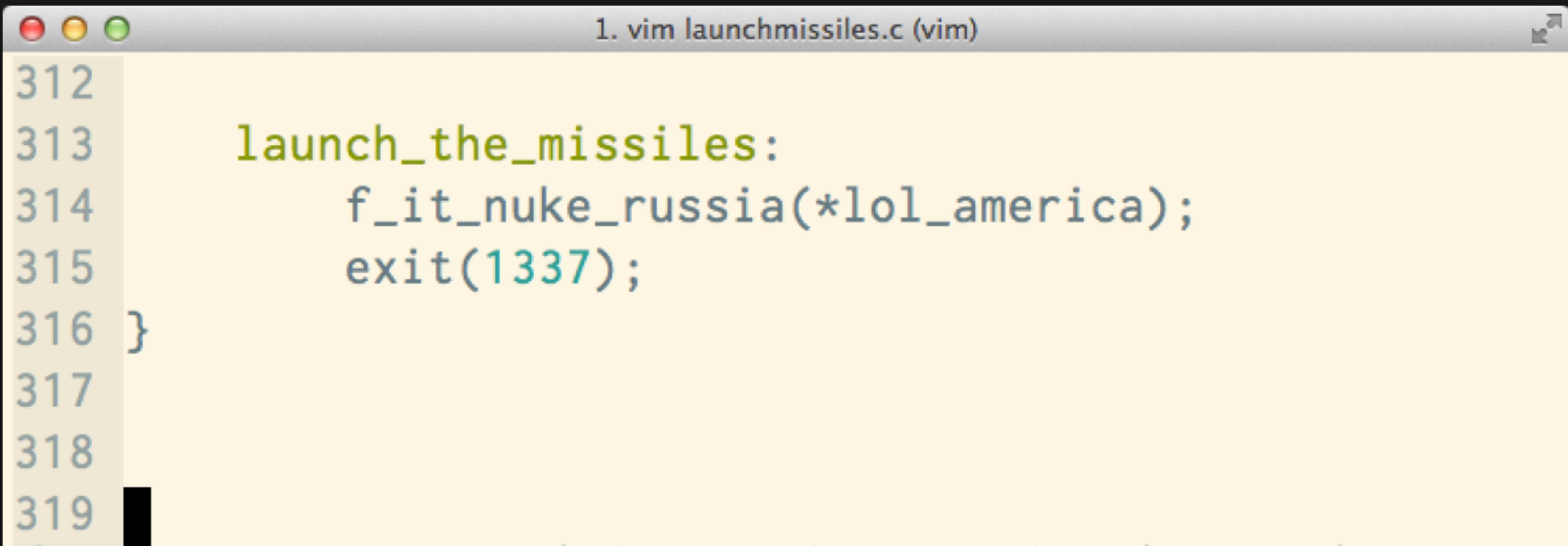
IMPERATIVE SUCKS ‘CAUSE MUTABILITY?



1. vim CollatzMain.java (vim)

```
0     public int get() { return value; }
1     public void set(int value) { this.value = value; }
2 }
3     public class CollatzThread extends Thread {
4         CurrentCollatz cc;
5         public CollatzThread(CurrentCollatz cc) { this.cc = cc; }
6         public void run(){
7             synchronized(cc) {
8                 int derp = cc.get();
9                 cc = new CurrentCollatz();
10                cc.set(derp);
11                (derp % 2 == 0) ? cc.set(derp / 2) : cc.set(derp * 3 + 1);
12            }
13        }
14    }
```

IMPERATIVE SUCKS ‘CAUSE GOTO?

A screenshot of a Mac OS X application window titled "1. vim launchmissiles.c (vim)". The window contains a single line of C code:

```
312
313     launch_the_missiles:
314         f_it_nuke_russia(*lol_america);
315         exit(1337);
316     }
317
318
319
```

The code defines a function named "launch_the_missiles" that calls "f_it_nuke_russia" with a pointer to "lol_america" and exits with the status code 1337. The code is numbered from 312 to 319 on the left side.

WHY DOES IMPERATIVE SUCK?

Loading and unloading instructions in registers
is done *procedurally*

(yes, I'm talking about you, assembly)

**OO AND FUNCTIONAL
ARE BROS**



Object-oriented programming is an
exceptionally bad idea which could
only have originated in California.

Edsger Dijkstra



The phrase ‘object-oriented’
means a lot of things. Half are obvious,
and the other half are mistakes.

Paul Graham



State is the root of all evil.
In particular, functions with side effects
should be avoided.

Joe Armstrong



Everything Object Oriented
Programming can do can be done
better in functional programming.

Doug Ransom



lol wat you mean ur code uses
a for loop in a single thread?

some idiot on IRC





Objected oriented programming and
functional programming are orthogonal,
not contrary to one another.

Martin Odersky

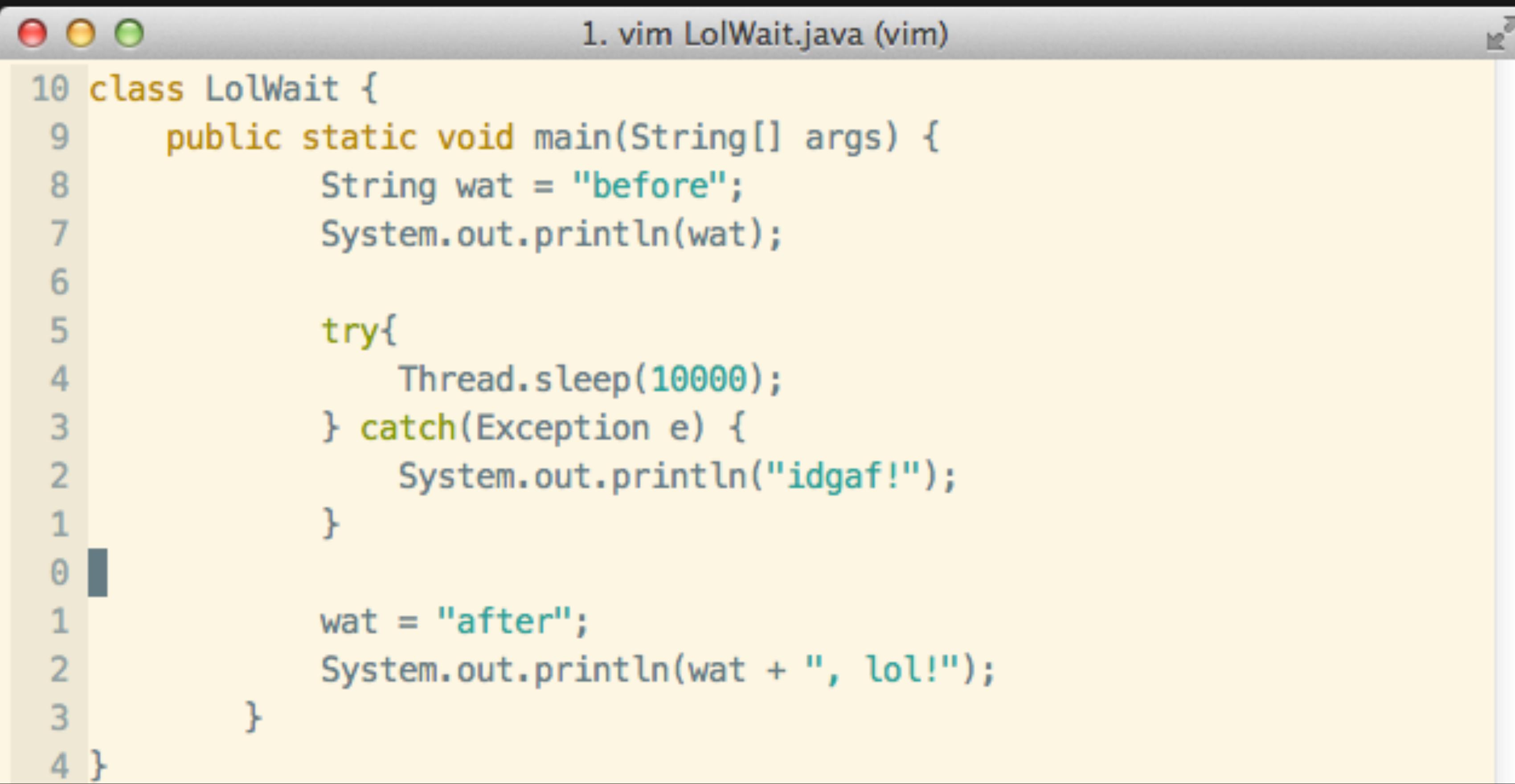
**THE INERRANT
FUNCTIONAL
DOGMA**



THE SACRED FUNCTIONAL VIRTUES

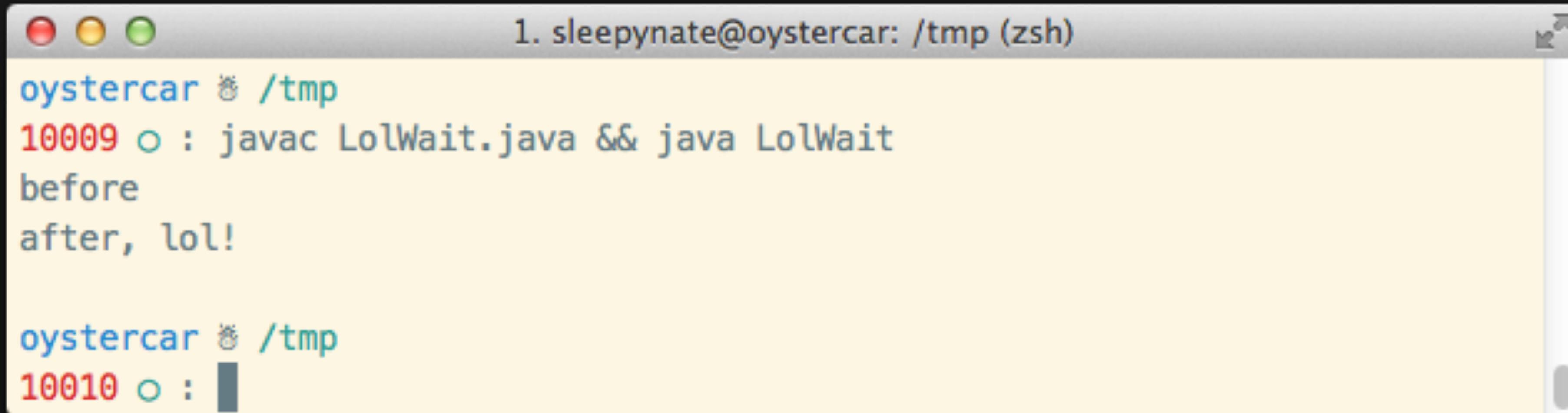
- Immutability is god
- First-class functions
- Composition and partial application

THERE IS NO GOD BEFORE IMMUTABILITY



```
1. vim LolWait.java (vim)
10 class LolWait {
9     public static void main(String[] args) {
8         String wat = "before";
7         System.out.println(wat);
6
5     try{
4         Thread.sleep(10000);
3     } catch(Exception e) {
2         System.out.println("idgaf!");
1     }
0
1         wat = "after";
2         System.out.println(wat + ", lol!");
3     }
4 }
```

THERE IS NO GOD BEFORE IMMUTABILITY



A screenshot of a terminal window with a light gray background and dark gray borders. The title bar reads "1. sleepynate@oystercar: /tmp (zsh)". The terminal output shows:

```
oystercar ~ /tmp
10009 o : javac LolWait.java && java LolWait
before
after, lol!

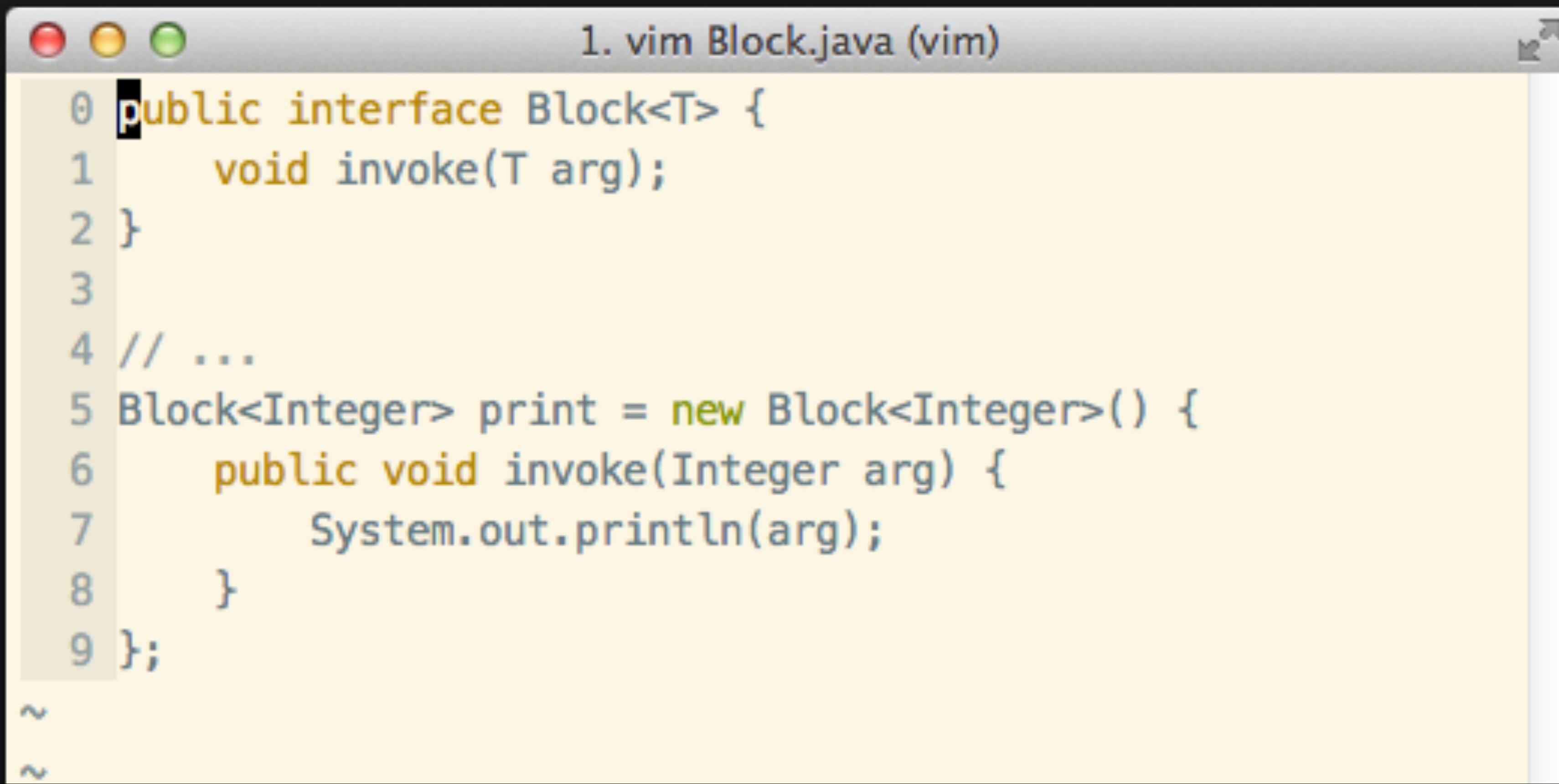
oystercar ~ /tmp
10010 o :
```

THERE IS NO GOD BEFORE IMMUTABILITY



```
0 import Control.Concurrent (threadDelay)
1 import Data.Time.Clock
2 import Data.IORef
3
4 sleep3 :: IO ()
5 sleep3 = do
6     threadDelay $ 1000000 * 3
7
8 main = do
9     wat <- newIORef "before"
10    wat' <- readIORef wat
11    putStrLn wat'
12
13    sleep3
14
15    writeIORef wat "after"
16    wat'' <- readIORef wat
17    putStrLn wat''
```

YOU CAN'T HAVE VERBS WITHOUT CLOSURES



A screenshot of a Mac OS X terminal window titled "1. vim Block.java (vim)". The window shows Java code for a "Block" interface and a "print" closure. The code is as follows:

```
0 public interface Block<T> {
1     void invoke(T arg);
2 }
3
4 // ...
5 Block<Integer> print = new Block<Integer>() {
6     public void invoke(Integer arg) {
7         System.out.println(arg);
8     }
9 };
```

The terminal window has the standard OS X red, yellow, and green close buttons at the top left. The title bar reads "1. vim Block.java (vim)". The code is displayed in a light-colored text area with syntax highlighting.

TRUE POWER COMES FROM COMPOSING LOGIC

```
/**  
 * Function composition.  
 *  
 * @param f A function to compose with another.  
 * @param g A function to compose with another.  
 * @return A function that is the composition of the given arguments.  
 */  
  
public static <A, B, C> F<A, C> compose(final F<B, C> f, final F<A, B> g) {  
    return new F<A, C>() {  
        public C f(final A a) {  
            return f.f(g.f(a));  
        }  
    };  
}
```

MY VOCABULARY SAYS YOU'RE DUMB



A monad is just
a monoid
in the category of
endofunctors
what's the problem?

FUNCTIONAL S*!T-TALKING

- “A monad is just a monoid in the category of endofunctors.”
- “Applicative Functors”
- Currying
- Zygomorphic Prepromorphisms ... stfu, no

THREE MONAD LAWS

Left identity: $\text{return } a \gg= f \equiv f a$

Right identity: $m \gg= \text{return} \equiv m$

Associativity: $(m \gg= f) \gg= g \equiv m \gg= (\lambda x \rightarrow f x \gg= g)$

THREE MONAD LAWS

- `do { f x }` is equivalent to `do { v <- return x; f v }`
- `do { m }` is equivalent to `do { v <- m; return v }`
- `do { x <- m; y <- f x; g y }` is equivalent to `do { y <- do { x <- m; f x }; g y }`

THREE MONAD LAWS

Monad(x).bind(f) is the same damn thing as f(x)

A screenshot of a Mac OS X desktop showing a vim window titled "1. vim Block.java (vim)". The window contains Java code demonstrating a monad. The code defines a public interface "Block<T>" with a single method "invoke(T arg)". It then creates a concrete implementation "add3" that adds 3 to its argument. Finally, it shows the expression "Monad(5).bind(add3);".

```
0 public interface Block<T> {
1     void invoke(T arg);
2 }
3
4 // ...
5 Block<Integer> add3 = new Block<Integer>() {
6     public Integer invoke(Integer arg) {
7         return arg + 3;
8     }
9 };
10
11 Monad(5).bind(add3);
12 // 8
```

THREE MONAD LAWS

Monad(x) is the same damn thing
as Monad(x)

THREE MONAD LAWS

$\text{Monad}(\text{Monad}(x).\text{bind}(f)).\text{bind}(g)$

is the same damn thing as

$\text{Monad}(f(x)).\text{bind}(g)$

THREE MONAD LAWS

- You don't tell noobs about Monads.
- You don't tell noobs about Monads.
- If it's your first Monad, you don't write a burrito tutorial.

THE MONAD LAW

“A monad is an object that
lets you use the object inside it”

FUNCTORS

“Stuff you can map over”

APPLICATIVE FUNCTORS

“Stuff you can map over,
and get a stuff of functions back”

CURRYING

“A function that takes an argument
and gives you a function that does the
same thing with one less argument”

ZYGOHISTOMORPHIC PREPROMORPHISMS

“Go home haskell, you’re drunk.”

A CRIB SHEET



**DECLARE,
DON'T ASSIGN**

**CALCULATE,
DON'T MUTATE**

**PATTERN MATCH,
DON'T IF-THEN-ELSE**

**RECUR,
DON'T LOOP**

**PASS FUNCTIONS,
NOT STRUCTS**

**MAP, FILTER,
AND REDUCE**

**DON'T BE
A BULLY**



Rank	Language	# Repositories Created
1	JavaScript	264131
2	Ruby	218812
3	Java	157618
4	PHP	114384
5	Python	95002
6	C++	78327
7	C	67706
8	Objective-C	36344
9	C#	32170
10	Shell	28561
11	CSS	17813
12	Perl	15412
13	CoffeeScript	11133
14	VimL	7857
15	Scala	6918
16	Go	6884
17	Prolog	5829
18	Clojure	4904
19	Haskell	4681
20	Lua	4048



Sure you can, it's pretty much like
what you do already

You