MY RUST IS SHIT!

A rant about rust idioms that make me go :-(

makemake

:333

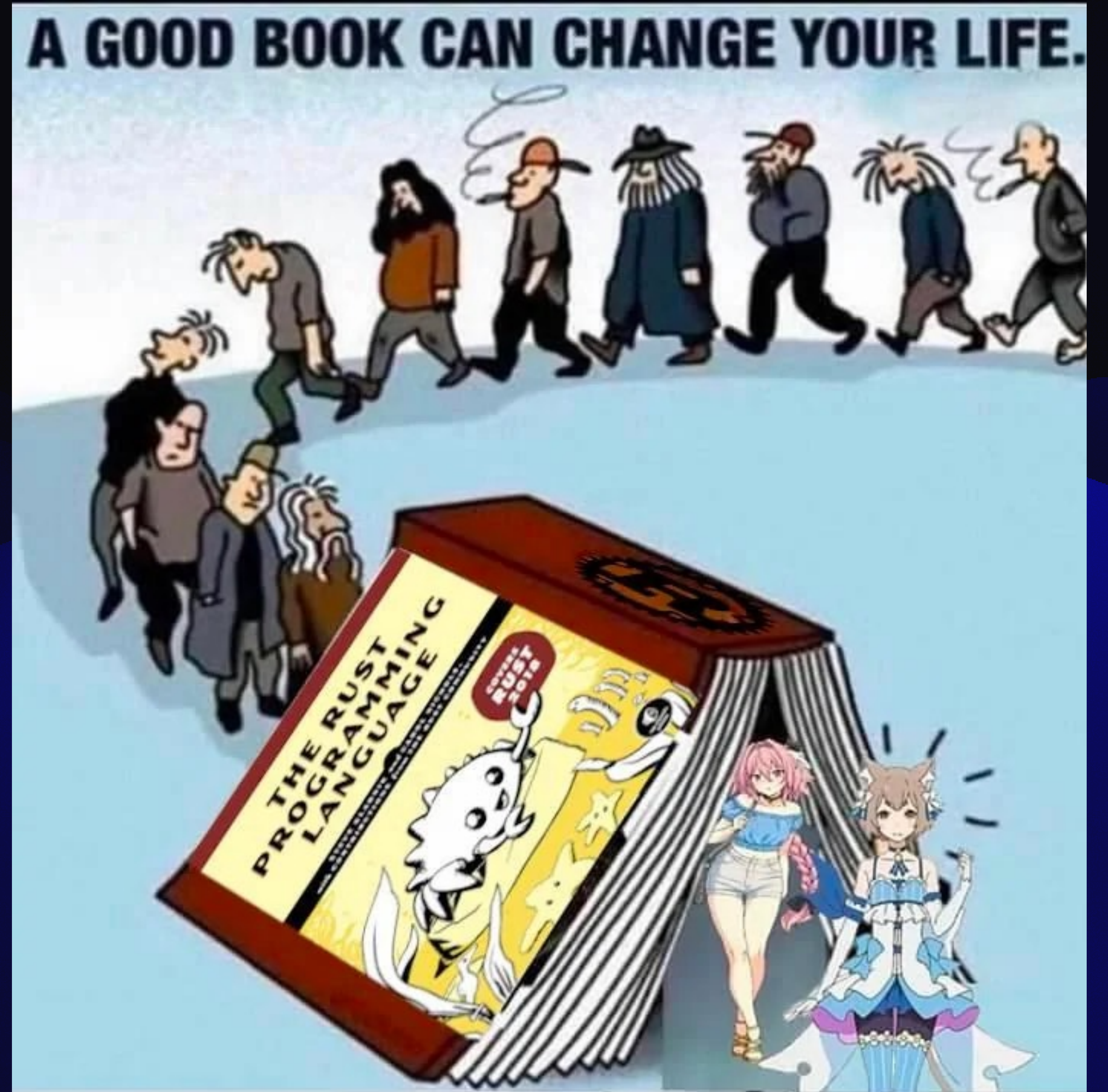
- Hiiiii my name is **makemake** and I write code :33<
- Most of you know how to code, but do you know how to *write code*?
- We’re going to be doing some introspection and look at rust coding conventions and challenge them.
- This is a nice way of me saying that this is essentially a group therapy session about bad rust code practices I personally dislike.
- Enjoy :D

THIS IS NOT ABOUT "CLEAN CODE"

Clean code is a psyop to make you write dog water code.

Theres nuance to everything!

What is *rustic*?



Do things the rust way

- Being rustic is about doing things in a rust specific way.
- Think about how you write and develop rust vs c/c++/whatever…
- You have:
  - Cargo, traits, borrowing, combinators, package management, no OOP slop, macros, etc…

Crab like thing on the bottom from Metroid this image is very relevant to rust idk what you mean

Rust is really good!

- Rust flexible, rust good.
- This is a double edged sword as it ends up being abused.
- Rust is imperative, and IMO should be written like an extension to C.

How 2 rust (imo dont crucify me)

But first, a short view back to the past…

Borrow checker & memory safety

- The borrow checker exists!
- Memory safety is a thing!
- It prevents us from being stupid with memory.
- It influences how we write code!

```rust
fn main() {
    let mut numbers = vec![1, 2, 3, 4, 5];

    // Immutable iterator chain
    numbers.iter()
        .map(|x| x * 2)
        .for_each(|x| println!("Double: {}", x));

    // Mutable iterator
    numbers.iter_mut()
        .for_each(|x| *x *= 3);

    println!("After tripling: {:?}", numbers);
}
```

Lets look at a c example

- Look at how much heavy lifting we have to do to get a memory safe equivalent :O
- This is because of iterators!

```
#include <stdio.h>

int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int length = sizeof(numbers) / sizeof(numbers[0]);  // pray we get this right

    // "Safe" iteration (trust me bro)
    for(int i = 0; i < length; i++) {  // hopefully i doesn't overflow
        printf("Double: %d\n", numbers[i] * 2);  // assumes multiplication doesn't overflow
    }

    // Modifying values (what could go wrong?)
    for(int i = 0; i < length; i++) {  // fingers crossed we don't go out of bounds
        numbers[i] *= 3;  // more overflow possibilities, wheee
    }

    printf("After tripling: [");
    for(int i = 0; i < length; i++) {
        printf("%d%s", numbers[i], i < length - 1 ? ", " : "");
    }
    printf("]\n");

    return 0;  // we made it! 🎉
}
```

Iterators

The best thing since sliced bread

- Iterators return the next item in an array and `None` when empty.
- THIS IS VERY POWERFUL!
- THIS IS ALSO HOW ASYNC WORKS!
- It allows us to write funny constructions with…

```rust
pub trait Iterator {
    type Item;

    // Required method
    fn next(&mut self) -> Option<Self::Item>;
}
```

Combinators

Or how I learned to start worrying about `.iter_into().map().zip().enumerate().filter().flat_map().chain().scan().take_while().collect().iter().for_each()` **during code review**

They’re cleaner!

They can be cleaner!

- Yes, they can, if you use them properly.
- Look at this, clean, simple, elegant!

```rust
fn main() {
    let raw_data = vec!["1", "2", "invalid", "4", "5", "bad", "7"];

    let sum: i32 = raw_data.iter()
        .filter_map(|s| s.parse::<i32>().ok())  // convert strings to numbers, skip invalid
        .filter(|&n| n > 3)                      // keep numbers > 3
        .fold(0, |acc, x| acc + x);             // sum them up

    println!("Sum of numbers > 3: {}", sum);  // prints: Sum of numbers > 3: 16
}
```

But they're often not…

- Do you have eyes???
- The code you see runs!!!
- I HAVE SEEN SIMILAR CODE IN PRODUCTION. SHIT LIKE THIS GETS MERGED!!!!

```rust
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    numbers.iter()
        .zip(numbers.iter().skip(1))  // pairs of adjacent numbers
        .map(|(a, b)| a + b)         // sum each pair
        .enumerate()                  // add indices
        .filter(|(i, x)| x % 2 == 0)  // only even sums
        .flat_map(|(i, x)| vec![i as i32, x].into_iter())  // flatten index and value
        .chain(std::iter::once(42))   // yeet a 42 at the end
        .scan(0, |state, x| {         // running total because why not
            *state += x;
            Some(*state)
        })
        .take_while(|&x| x < 100)     // stop before we hit 100
        .collect::<Vec<_>>()          // collect into vector
        .iter()
        .for_each(|x| println!("😱 {}", x));  // print with emoji because we're fancy
}
```

They’re faster!

What!?

How? Why? Hello??

Compilers are very powerful

- The closest we have to AGI are modern compilers.
- They optimize pretty much everything that doesn’t change your code semantics (unnecessary clones will still happen!)
- Combinators have slightly different semantics but its mostly irrelevant here.

Do not fall for the `ffmpeg` propaganda!

You can not write better assembly than the compiler.

Affirm.

They’re rustic!

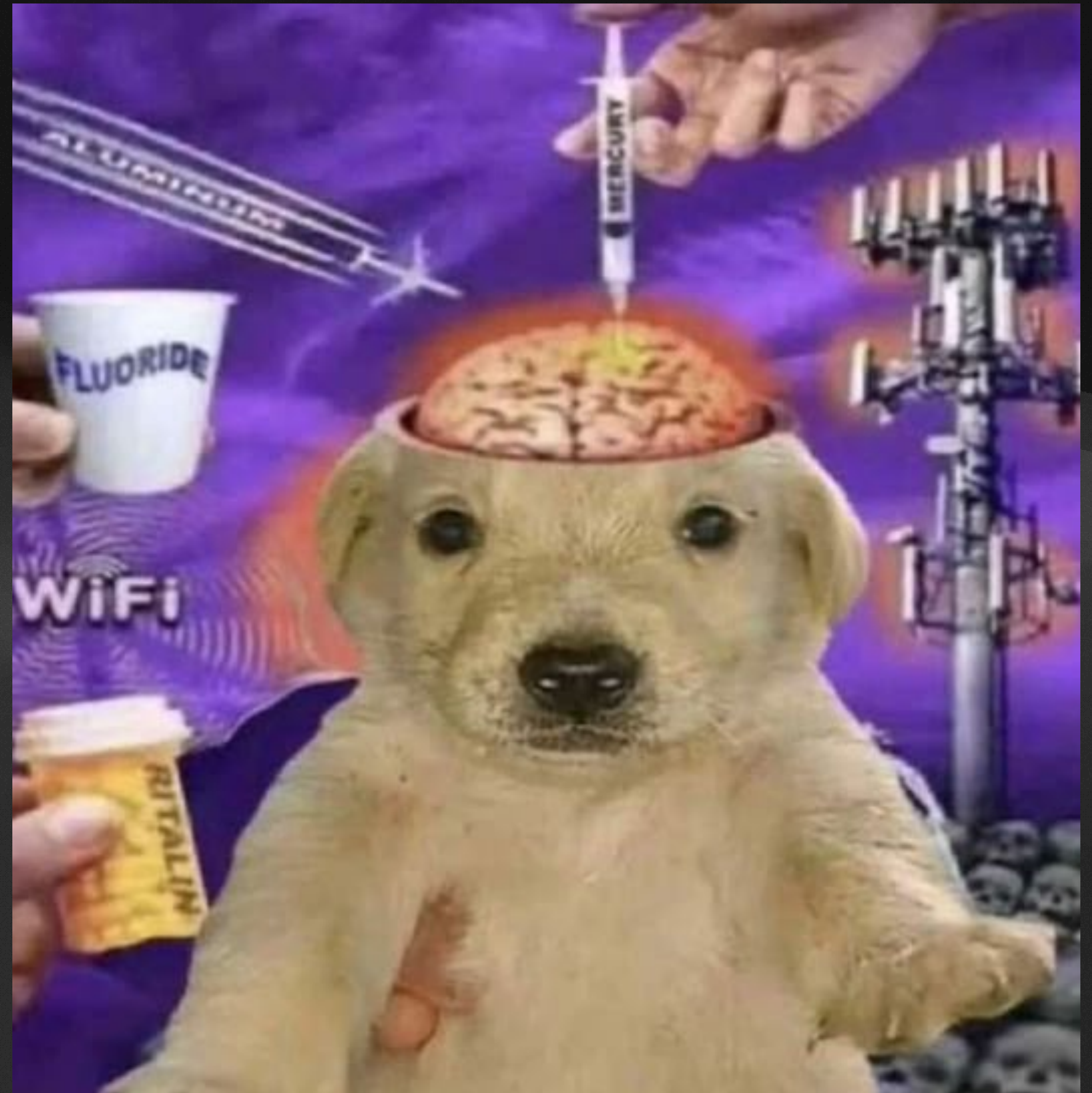
Is rustic code always good?

- No!
- You shouldn’t write code that is bad, slow, or illegible in a rust way when you can do it better in a non rust way.

```rust
// Find most frequent number and its neighbors - basic edition
fn simple_way(nums: &[i32]) -> Option<(i32, Vec<i32>)> {
    let mut max_count = 0;
    let mut most_common = None;

    // Find the most common number
    for &num in nums {
        let count = nums.iter().filter(|&&x| x == num).count();
        if count > max_count {
            max_count = count;
            most_common = Some(num);
        }
    }

    // Get its neighbors
    most_common.map(|n| {
        let neighbors = nums.windows(3)
            .filter(|w| w[1] == n)
            .map(|w| vec![w[0], w[2]])
            .next()
            .unwrap_or_default();
        (n, neighbors)
    })
}

// Same thing but we choose CHAOS
fn spicy_way(nums: &[i32]) -> Option<(i32, Vec<i32>)> {
    nums.iter()
        .copied()
        .zip(
            std::iter::repeat(nums)
                .take(nums.len())
        )
        .map(|(num, slice)| {
            (num, slice.iter()
                    .filter(|&&x| x == num)
                    .count())
        })
        .fold(
            std::collections::HashMap::<i32, usize>::new(),
            |mut acc, (num, count)| {
                acc.entry(num)
                    .and_modify(|e| *e = (*e).max(count))
                    .or_insert(count);
                acc
            }
        )
        .into_iter()
        .max_by_key(|(_, count)| *count)
        .and_then(|(num, _)| {
            nums.windows(3)
                .filter(|w| w[1] == num)
                .next()
                .map(|w| (
                    num,
                    vec![w[0], w[2]]
                ))
        })
}
```

When should you not use combinators?

- Your looping has side effects.
- You need to handle non trivial errors and propagate them upwards
- You have to use really long combinator chains for your desired effects.
- When they look weird :/



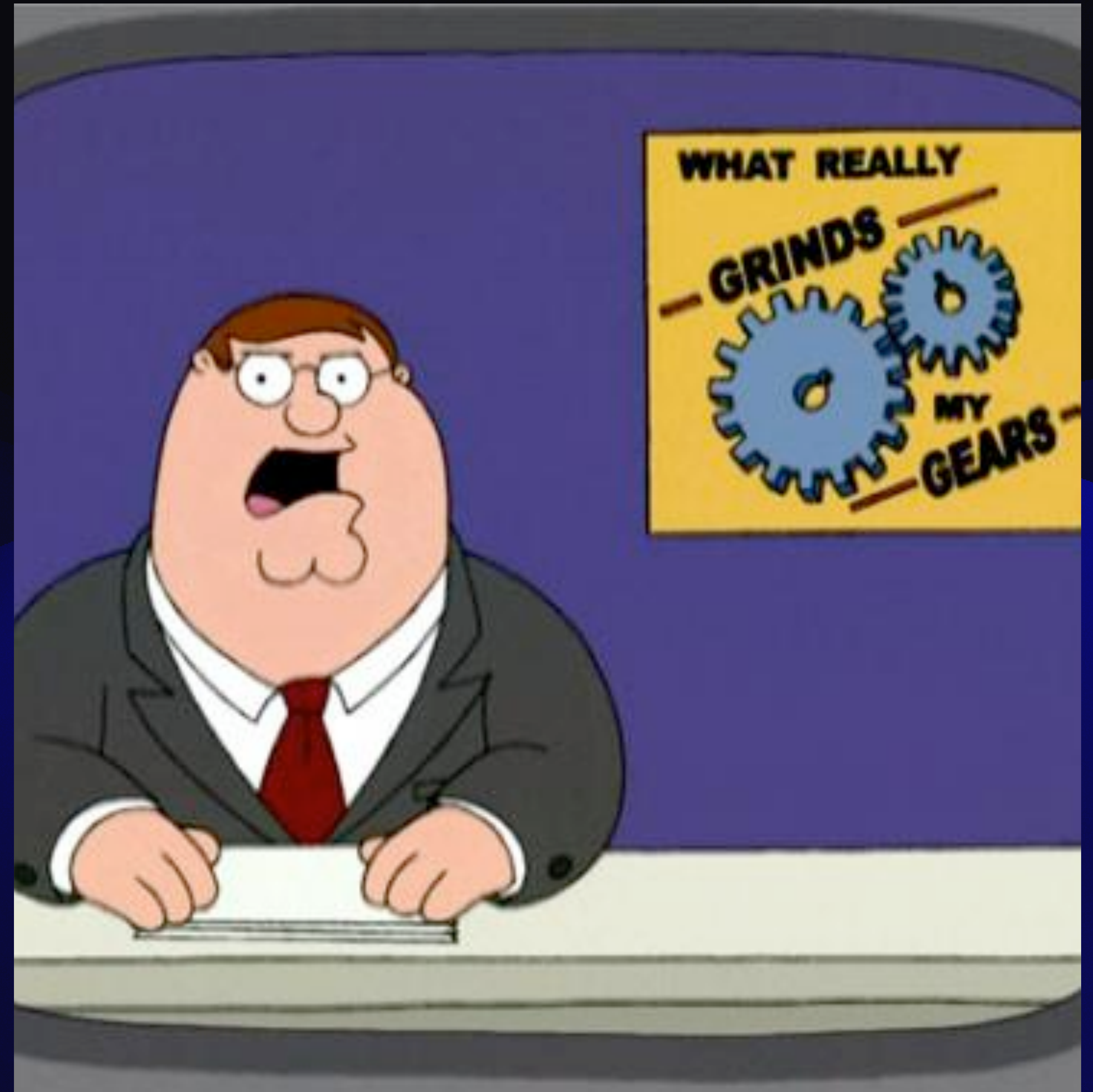
Attention grabbing image

Wen combinator ser?

- Code that only affects the thing ur directly interacting with.
- Small interaction chains that do a “one-liner”.
- Combinator chains get exponentially hard to reason about when you start adding more of them.
- Remember, this is just my opinion. Go wild.

You know what grinds my gears?

When people write rust like java.

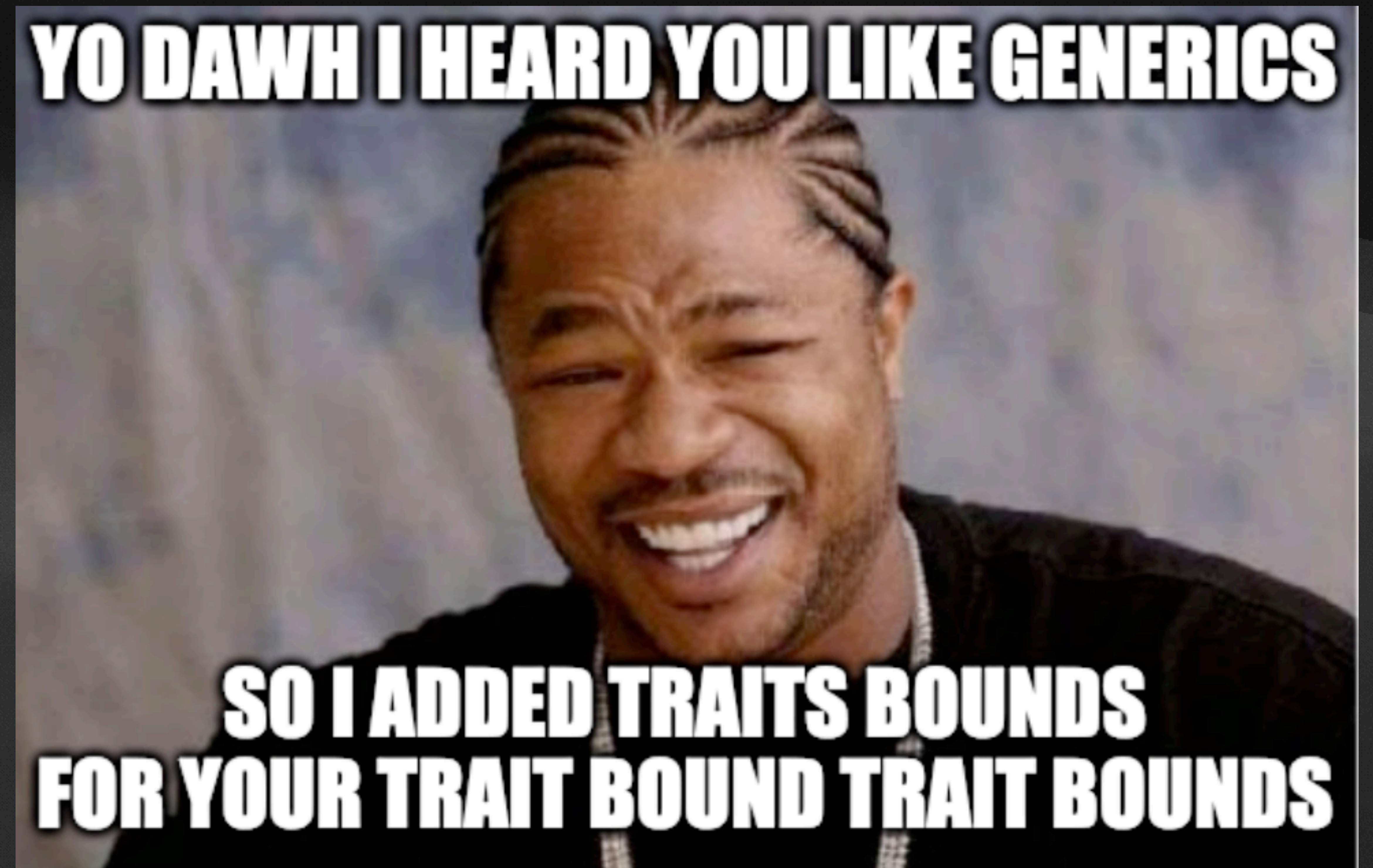


Write once, run away

- Using Java for serious jobs is like trying to take the skin off a rice pudding wearing boxing gloves.
- If you think applying java programming idioms to other languages is a good idea, please stop programming.

Traits != interfaces

- Java interfaces and traits are similar in functionality.
- Rust is not OOP!
- Traits are just markers to tell the compiler about function signatures and what u need for the output.
- You will pay the price of trying to make everything generic eventually…



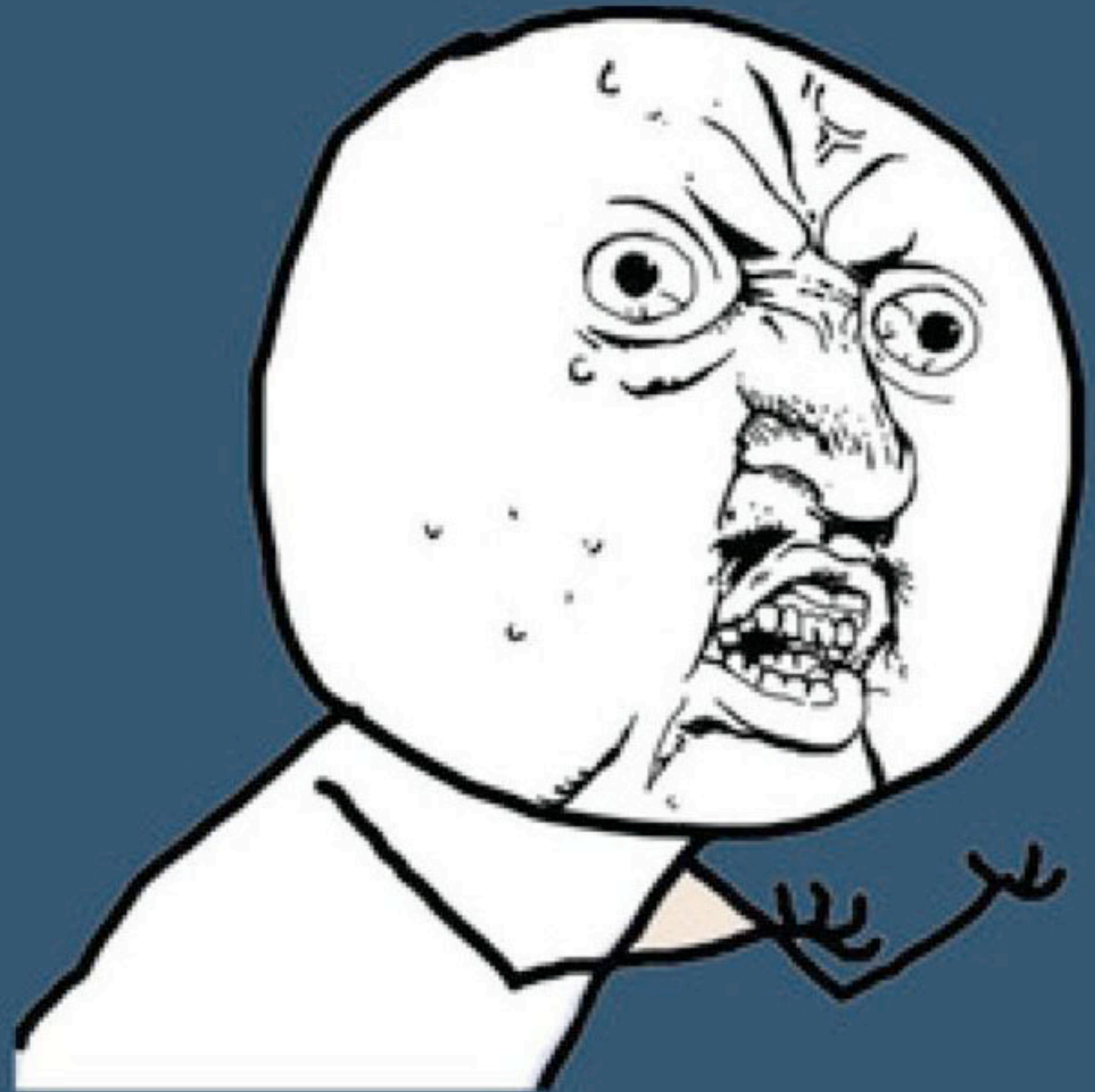
Trait fuckery

- To see how traits can go wrong, lets implement some!
- We're going to implement a basic trait for some object!

```
trait Description {
    fn describe(&self) -> String;
}
```

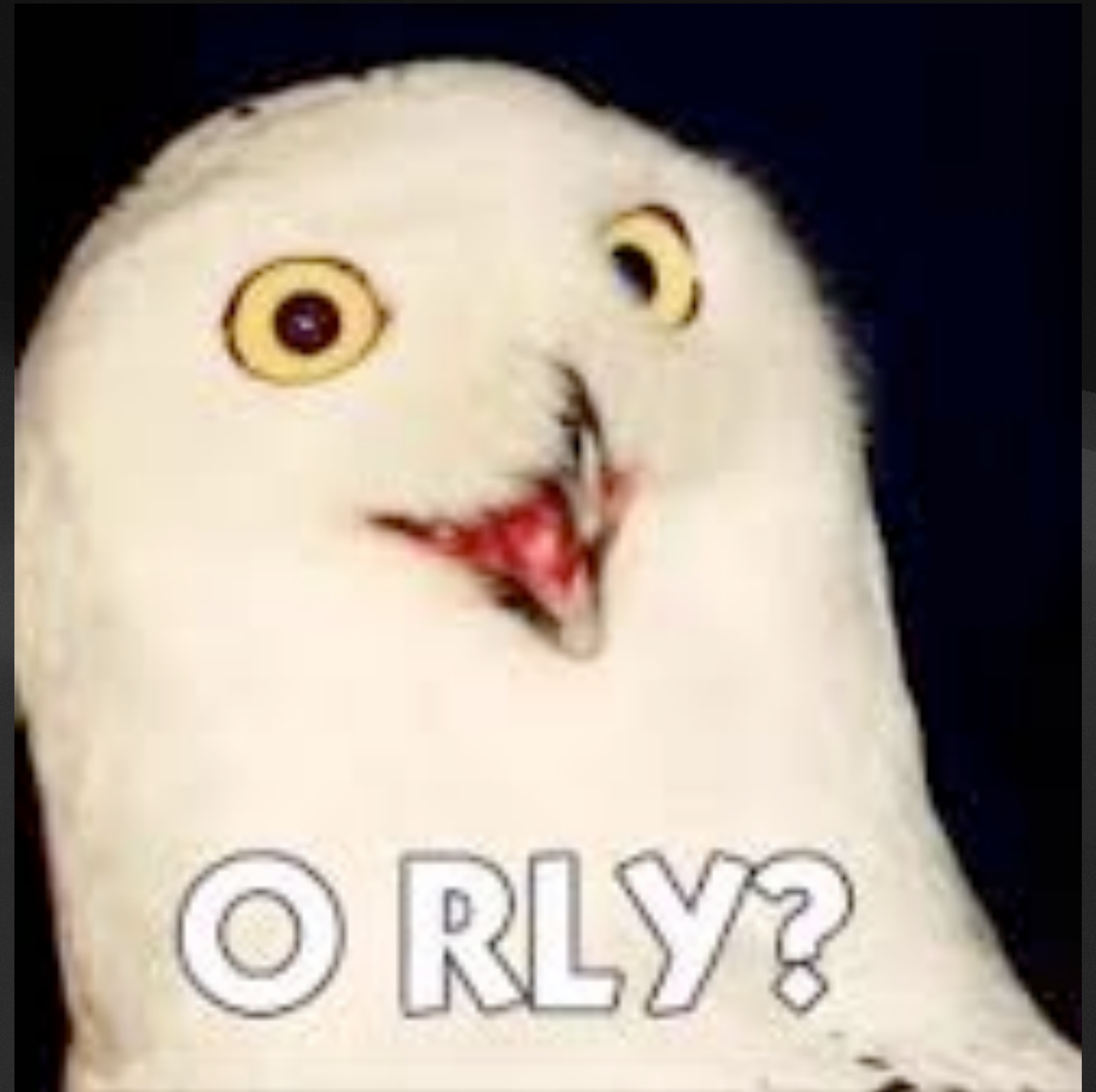
Womp womp

```rust
trait Named {
    fn name(&self) -> String;
}

struct Service {
    named: Named
}
```



Wat do???

```rust
trait Description {
    fn describe(&self) -> String;
}

struct Thing {
    obj: Box<dyn Description>,
}
```



The heap makes me sad

- The heap is slow and we want to avoid it.
- Thankfully we can if we specify the types of everything at compile time.
- If everything is sized we don't need to box. Neat.

```
trait Description {
    fn describe(&self) -> String;
}

struct Thing<T>
where
    T: Description
{
    obj: T,
}
```

Shits fugged m8

```rust
// Implementation with even more bounds
impl<'a, T, U> Thing<'a, T, U>
where
    T: Description + std::fmt::Display,
    U: AsRef<str> + std::fmt::Debug,
{
    fn new(obj: &'a T, label: &'a U) -> Self {
        Self {
            obj,
            label,
            tags: Vec::new(),
        }
    }

    fn with_tag(mut self, tag: &'a str) -> Self {
        self.tags.push(tag);
        self
    }

    fn describe_all<V>(&self, other: &'a V) -> String
    where
        V: Description + std::fmt::Debug,
    {
        format!(
            "Thing labeled {:?} with tags {:?}:\n{}\nCompared to:\n{}",
            self.label.as_ref(),
            self.tags,
            self.obj.describe(),
            self.obj.compare(other)
        )
    }
}
```

```rust
#[derive(Debug, Clone)]
struct Cat {
    name: String,
}

impl Description for Cat {
    fn describe(&self) -> String {
        format!("Cat named {}", self.name)
    }

    fn compare<'a, U>(&'a self, other: &'a U) -> String
    where
        U: Description + 'a,
    {
        format!("{} vs {:?}", self.name, other)
    }
}

impl std::fmt::Display for Cat {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "🐱 {}", self.name)
    }
}

fn main() {
    let cat = Cat { name: "Whiskers".to_string() };
    let cat2 = Cat { name: "Mittens".to_string() };
    let label = "my-cat";

    let thing = Thing::new(&cat, &label)
        .with_tag("cute")
        .with_tag("fluffy");

    println!("{}", thing.describe_all(&cat2));
}
```

Is there a cure?

“Sometimes the clean, elegant implementation is just a function. Not a method. Not a class. Not a framework. Just a function.”

- John Carmack, legendary programmer and founder of id software

Thank you!

Thoughts? Opinions? Questions?

Blog with all my socials!

Twitter