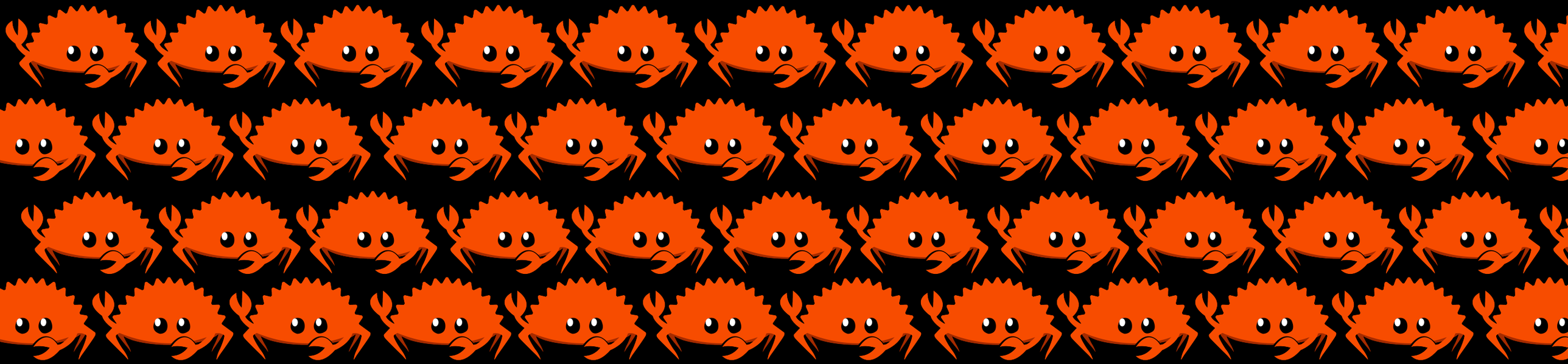


The Beauty of Rust

Structs

Peter Zdankin

Object Orientation

What, why and how

- Whenever you code, you quickly encounter that data is linked to other data
- “I have a customer that has a customer ID and a name”
- It is a good idea to group this data to objects that all have instances of this linked data
  - You don’t mix up whose id belongs to which name
  - You always have a name and an id
- There are some approaches how to implement this

Two competing approaches

Inheritance

- X is a Y
- Base classes that get ever more complex
- Specification
- Share data, behaviour and implementation

Composition

- X has a Y
- Modular approach
- Interchangeable elements in each struct

Classes

- You are coding a dog
- Dog
  .bark()

Classes

- You also need a cat
- Dog
  .bark()
- Cat
  .meow()

Classes

- Both poop
- Dog
  .bark()
  .poop()
- Cat
  .meow()
  .poop()

Classes

- Looks like we can extract this
- Animal
  .poop()
  Dog
    .bark()
  Cat
    .meow()

Classes

- You need a cleaning robot for said poop
- Animal
  .poop()
  Dog
    .bark()
  Cat
    .meow()
- CleaningRobot
  .drive()
  .clean()

Classes

- You also need a murder robot
- Animal
  .poop()
  Dog
    .bark()
  Cat
    .meow()
- CleaningRobot
  .drive()
  .clean()
- MurderRobot
  .drive()
  .kill()

Classes

- Extract!
- Animal
  .poop()
  Dog
    .bark()
  Cat
    .meow()
- Robot
  .drive()
  CleaningRobot
    .clean()
  MurderRobot
    .kill()

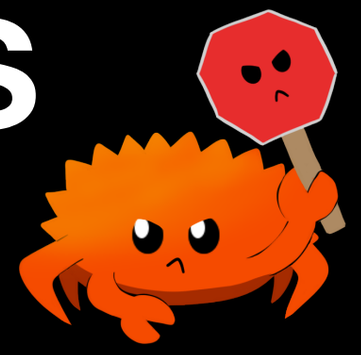
The IT team feeling the disaster

Regional Manager:
”YO YO YO WASSUP MY HOMIE”

“We need two new things. A MurderDogRobot that must bark, kill and drive, but cannot poop. We also need KillerCat that kills and poops, but is no robot”

“That should be easy right?”

The IT team after this new task



Instead, composition

- Dog = compose(Barker, Pooper)
- Cat = compose(Meower, Pooper)
- KillerCat = compose(Killer, Meower, Pooper)
- CleaningRobot = compose(Driver, Cleaner)
- MurderRobot = compose(Driver, Killer)
- MurderRobotDog = compose(Driver, Killer, Barker)

Composition

Advantages, lots of advantages

- You are flexible in your object design
- You can quickly exchange objects in other objects
- You can easily mock

```rust
struct Pooper();

impl Pooper{
    fn poop(&self) {
        println!("💩")
    }
}

struct Meower();

impl Meower{
    fn meow(&self) {
        println!("MEOW")
    }
}
```

```rust
struct Cat{
    pooper: Pooper,
    meower: Meower
}

impl Cat{
    fn new() -> Cat{
        Cat{pooper: Pooper{}, meower: Meower{}}
    }
}

fn main() {
    let cat = Cat::new();
    cat.pooper.poop();
    cat.meower.meow();
}
```

Rust Structs

Deeper dive

- You first write a struct
- That struct contains all data, which the struct needs to store

- You then impl(ement) methods and functions associated with the struct
- Methods use either &self or &mut self to indicate that they access member variables
- Functions operate on no object, and hence have only their parameters

```
struct Gun{
    ammo_in_clip: u32,
    clip_size: u32,
    total_ammo: u32,
    damage_per_hit: u32,
}
```

```
impl Gun{
    fn new() -> Gun{
        Gun{ammo_in_clip: 12, clip_size: 12, total_ammo: 120, damage_per_hit: 10}
    }

    fn reload(&mut self){
        println!("RELOADING");
        if self.total_ammo > self.clip_size{
            self.ammo_in_clip = self.clip_size;
            self.total_ammo -= self.clip_size;
        }else{
            self.ammo_in_clip = self.total_ammo;
            self.total_ammo = 0;
        }
    }
```

```
    fn pew(&mut self){
        if self.must_reload(){
            self.reload();
        }
        if self.ammo_in_clip > 0{
            self.ammo_in_clip -= 1;
            println!("PEW {} damage", self.damage_per_hit);
        }
    }

    fn must_reload(&self) -> bool{
        self.ammo_in_clip == 0
    }
}
```

Something Fancy

- Imagine you want to print your struct for debug purposes
- Other languages: Write a to_string method (or something similar)
- Rust: Generate the code to print it and use the debug formatter

```rust
#[derive(Debug)]
struct Gun{
    ammo_in_clip: u32,
    clip_size: u32,
    total_ammo: u32,
    damage_per_hit: u32,
}
```

```rust
println!("{:?}",gun);
```

```
Gun { ammo_in_clip: 4, clip_size: 12, total_ammo: 108, damage_per_hit: 10 }
```

Exercises

- Do the MurderDogRobot Example in Rust
- Write a struct that holds rectangles and write a function, that tells you which struct is bigger
- Write a struct that holds an array of integers and has two methods to sort the numbers ascending and descending
- Try to borrow data out of a struct, or move ownership of data out of a struct
- Write two structs (e.g. Circle and Rectangle) and let them both have an area method. What do you observe?