A Brief Overview of Entity Component Systems with Rust and Bevy

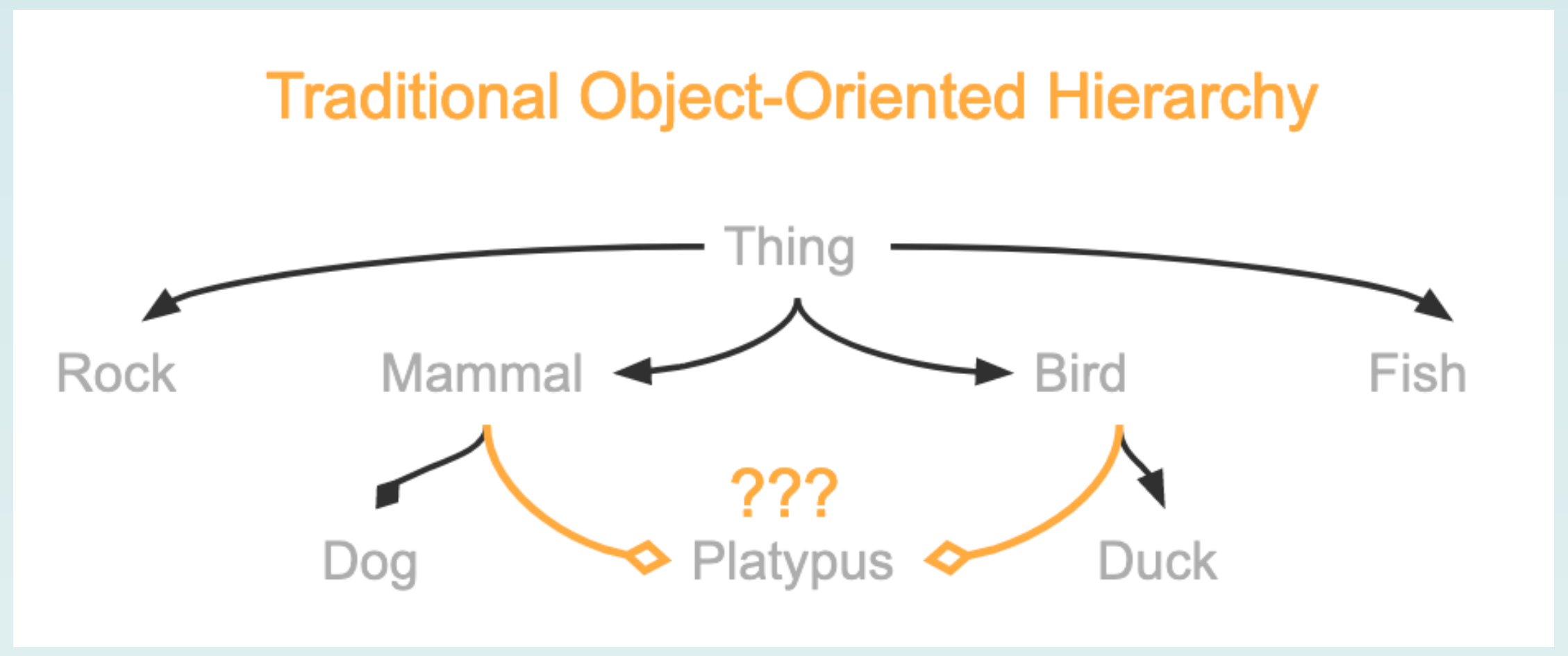
Carson Webster

UC SANTA CRUZ | Baskin Engineering

Motivation for Entity Component Systems

- Criticism of Java's Object Oriented Programming approach where one big root node class exists and other classes inherit from the root node functionality to create more distinct classes

- This becomes a problems when we want to create a class that needs to inherit functionality from multiple types of classes



How Entity Component Systems Work

- Conceptually, think of a table containing all the entities as rows and attached components as a boolean in a column

Entity ID	Position?	Damage?	on_ground?
0	✅	✅	✅
1	✅	❌	❌
2	✅	❌	✅
3	❌	❌	❌

- Ecs_table[2] = 101

- Like a big hash table, a good ECS will look up an entity ID # to find an integer than can be bit indexed to determine attached components.
- In this case, the bits associated with entity ID #2 indicate this entity has a position component, no damage component, and an on_ground component.

What is an Entity Component System?

- An architectural and data organization pattern using three parts:

Entities

- Represent all of our things in the program

- Referred to by an ID #

- Player, Items, Enemies, Particles, Walls, Anything!

Components

- Data we attach to our entities

- Structures that hold one or many data types

- Position (int x, int y)
  Damage (float)
  On_Ground(bool)

Systems

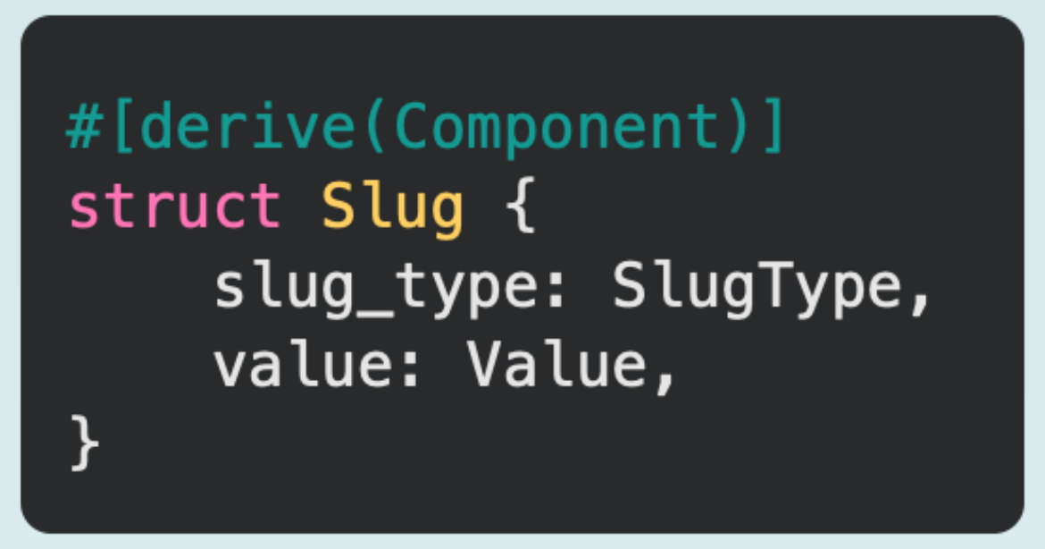
- Functions that are run to query and mutate components

- The logic of our program. Defines how an entity and attached components will change over time

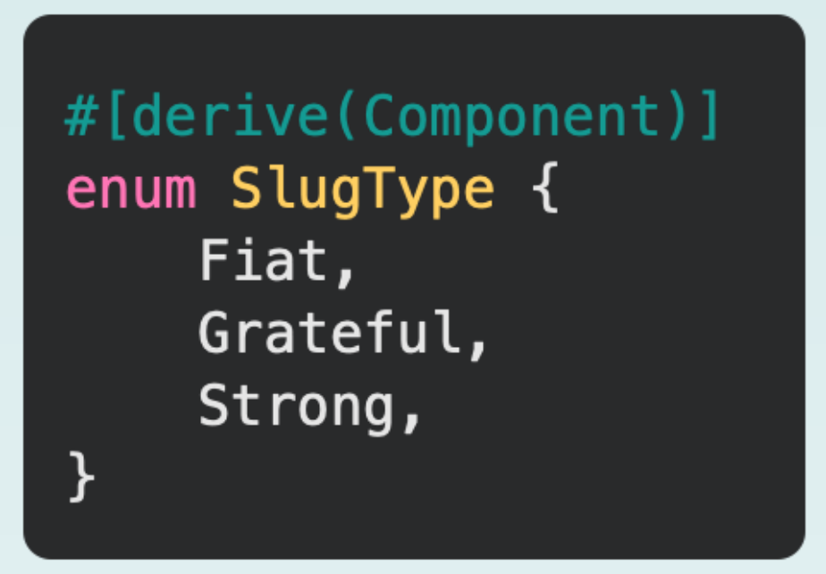
- function update_position(query<position>)

A small example program

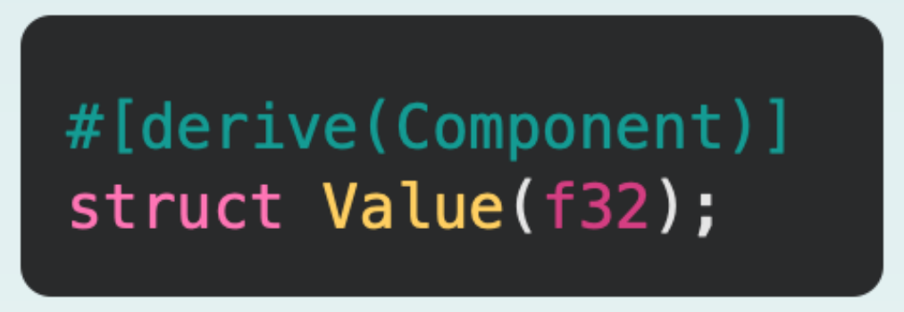
- Let's pretend we are writing a program that simulates pulling a random sticker from one of those quarter vending machines.
- In our vending machine exists three variants of Sammy the Slug we can receive
- Before each program run, spawn a Slug entity with a random variant
- During execution, print the random slug that was spawned



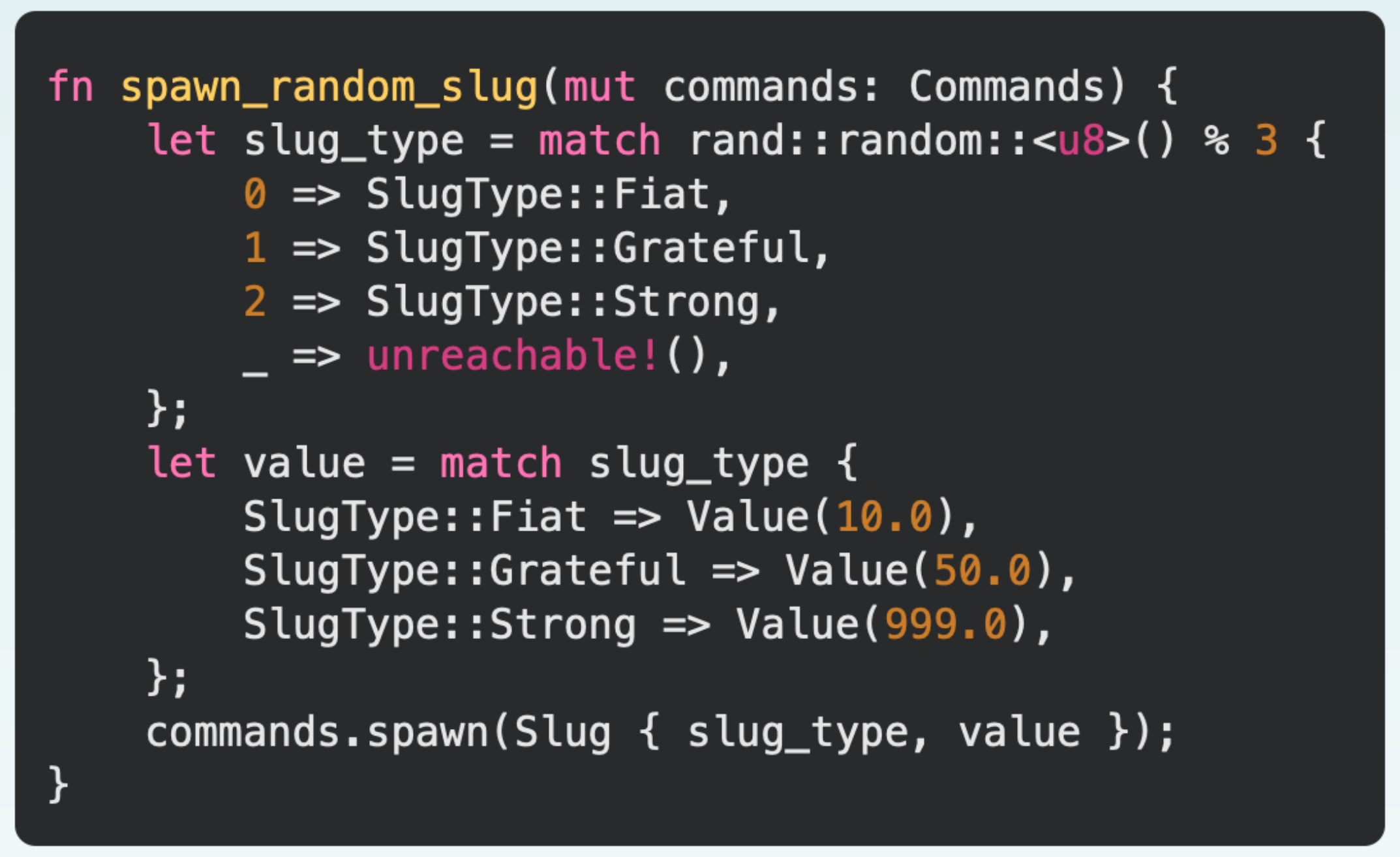
```rust
#[derive(Component)]
struct Slug {
    slug_type: SlugType,
    value: Value,
}
```



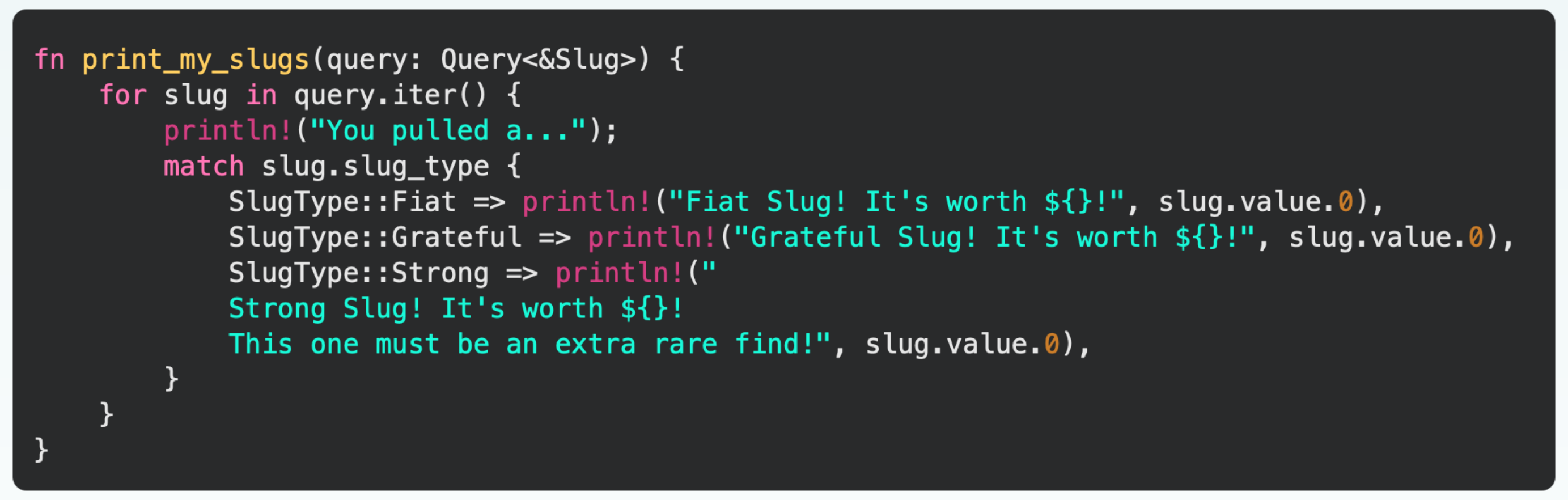
```rust
#[derive(Component)]
enum SlugType {
    Fiat,
    Grateful,
    Strong,
}
```



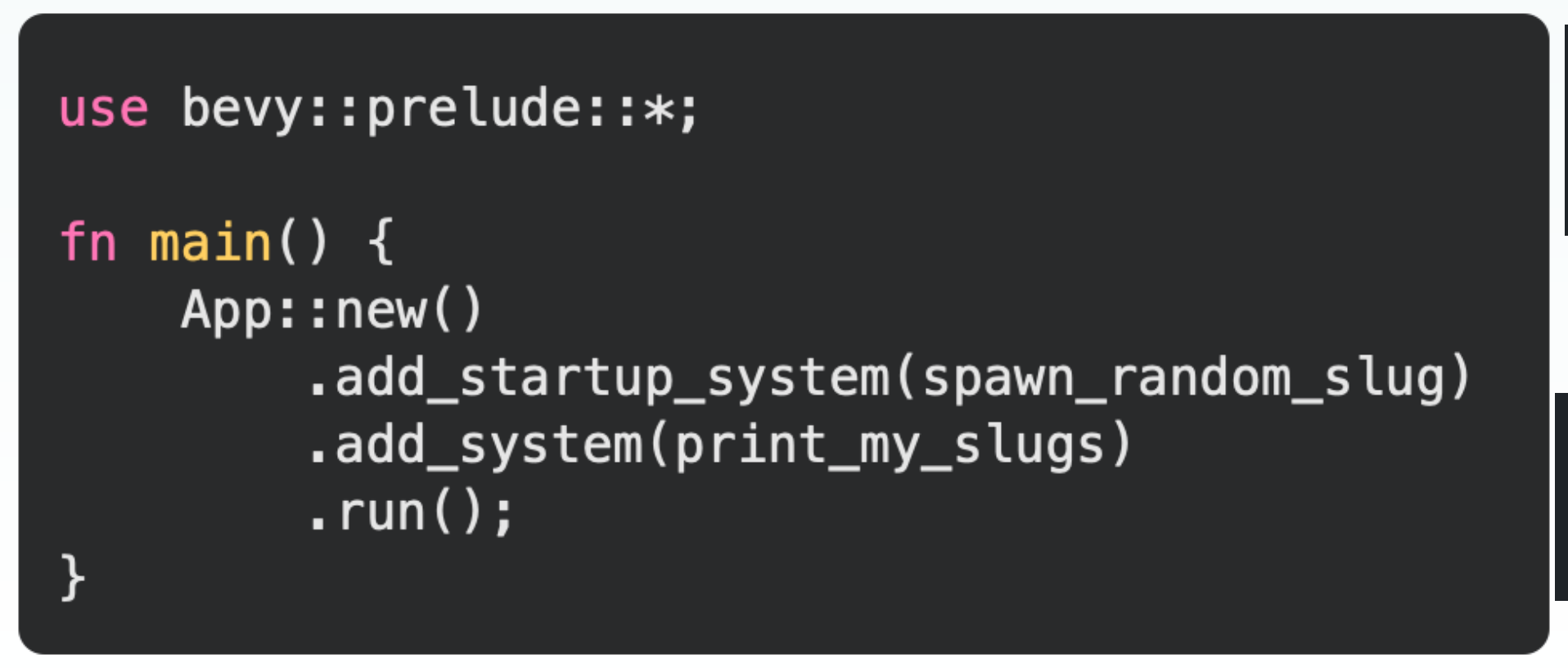
```rust
#[derive(Component)]
struct Value(f32);
```



```rust
fn spawn_random_slug(mut commands: Commands) {
    let slug_type = match rand::random::<u8>() % 3 {
        0 => SlugType::Fiat,
        1 => SlugType::Grateful,
        2 => SlugType::Strong,
        _ => unreachable!(),
    };
    let value = match slug_type {
        SlugType::Fiat => Value(10.0),
        SlugType::Grateful => Value(50.0),
        SlugType::Strong => Value(999.0),
    };
    commands.spawn(Slug { slug_type, value });
}
```

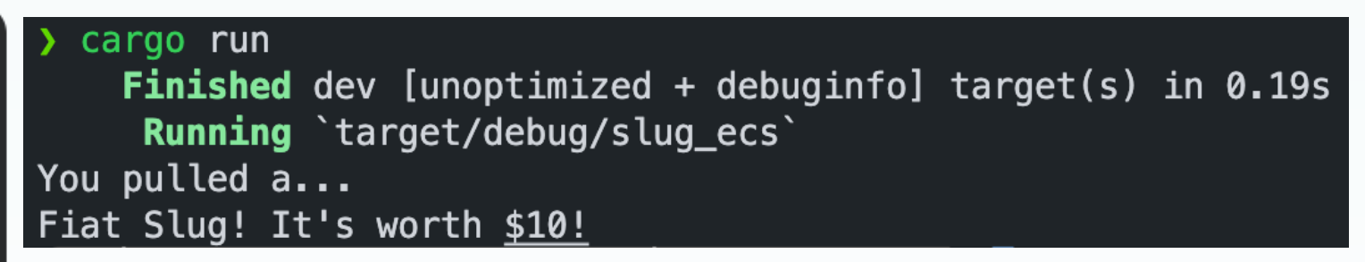


```rust
fn print_my_slugs(query: Query<&Slug>) {
    for slug in query.iter() {
        println!("You pulled a...");
        match slug.slug_type {
            SlugType::Fiat => println!("Fiat Slug! It's worth ${}!", slug.value.0),
            SlugType::Grateful => println!("Grateful Slug! It's worth ${}!", slug.value.0),
            SlugType::Strong => println!("
            Strong Slug! It's worth ${}!
            This one must be an extra rare find!", slug.value.0),
        }
    }
}
```

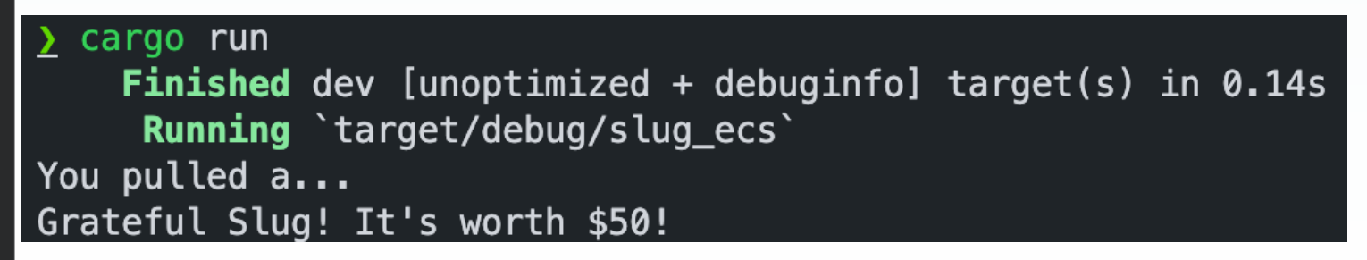


```rust
use bevy::prelude::*;

fn main() {
    App::new()
        .add_startup_system(spawn_random_slug)
        .add_system(print_my_slugs)
        .run();
}
```



```
❯ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.19s
     Running `target/debug/slug_ecs`
You pulled a...
Fiat Slug! It's worth $10!
```



```
❯ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.14s
     Running `target/debug/slug_ecs`
You pulled a...
Grateful Slug! It's worth $50!
```