Object Oriented Programming in Rust: Traits

CS 110L

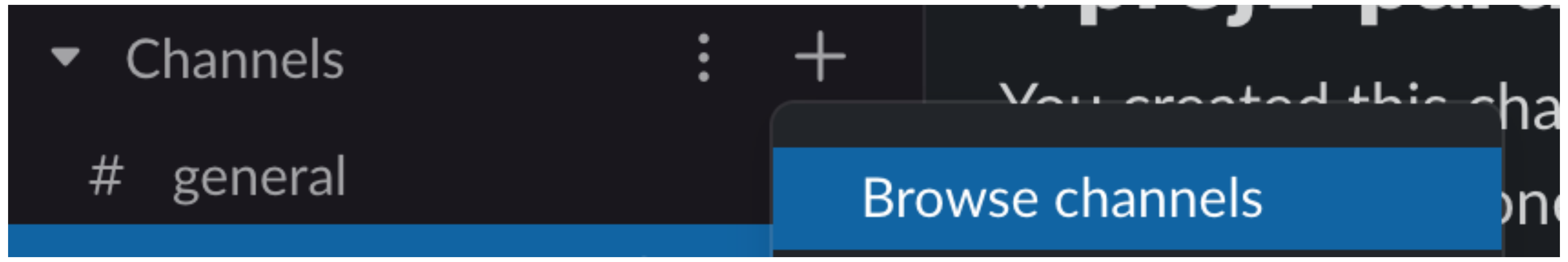
January 31, 2022

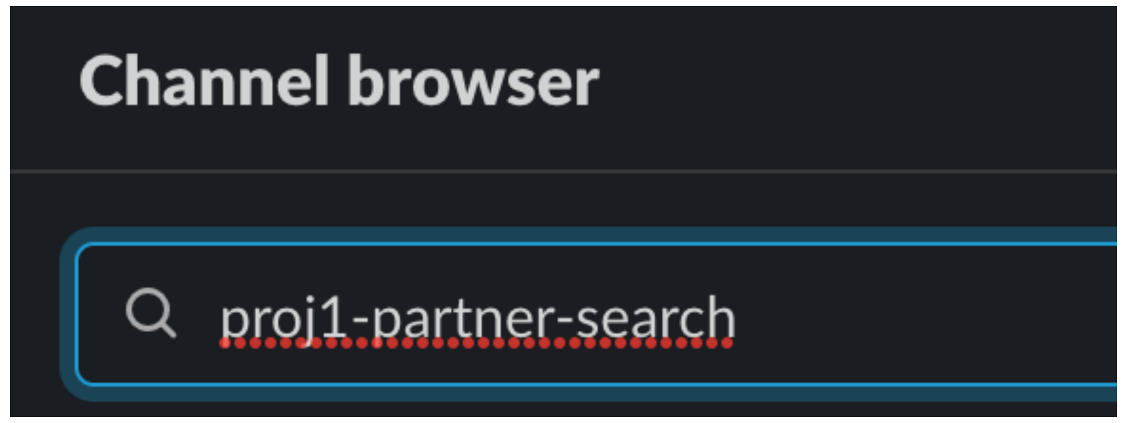
Logistics

- Reminder on participation:
  - Participation makes the class more engaging and effective for all of us
  - Part of the grade => incentivize you to think about and stay up-to-date with the material
  - Attending class — asking and answering questions, contributing to discussion, etc.
  - Slack — asking and answering questions, contributing to discussion, etc.
- Week 3 exercise were due! (Sample solutions released tomorrow.)
  - *These are the most challenging exercises that we'll do this quarter!*
  - *Thanks for attempting and sticking with it :)*

Logistics

- Project 1 out next week! Build a debugger!
- You can work alone or in groups of 2-3
- Find project partners in class, and feel free to post in #proj1-partner-search Slack channel
  - *Next to channels, click `+` -> Browse channels -> search for “proj1-partner-search”*





proj1-partner-search

- Before pairing up, communicate with each other: What are your goals & ideal outcomes for this project? How much time do you have to spend on it?

Object Oriented Programming in C++

Classes

- "Object" Oriented: Create an 'object' - movie database, and you can perform **methods** on this object.
- You can create **instances** of objects, and each would have their own set of variables. (Movie database with different files)
- Classes divided into **public** and **private** regions.
- **public** members can be accessible to anyone with reference to an instance
- **private** members only accessible to the implementer of the class

```
class imdb {
  public:
    imdb(const std::string& directory)
    bool getCredits(...)
  private:
    /* Elements
    const char* kActorFileName;
}
```

What are some advantages to Classes?

Advantages to Class Design

- **Modularity:** We can break down a big system into manageable components that provide clear interfaces and can be tested in isolation.
- **Encapsulation**: Group related data and methods together into a single “object.”
- **Code-Hiding**: Don't need to expose parts of a class not needed for a user to interact with it.
- **Code-Reuse:** Want an object to be different based on the file it takes in? Add one parameter to its constructor, and suddenly you have two different implementations, but just one class!
- *Other things? What do you think?*

Reusing code with “inheritance”

A bunch of slightly different types of teddy bears = lots of repeated code!

```
class TeddyBear {
  public:
    TeddyBear(..);
    void roar_sound();
}
```

```
class PurpleTeddyBear {
  public:
    TeddyBear(..);
    void roar_sound();
    void purple_button_song();
}
```

```
class RedTeddyBear {
  public:
    TeddyBear(..);
    void roar_sound();
    void red_button_song();
}
```

```
class PurpleTeddyBear {
  public:
    TeddyBear(..);
    void roar_sound();
    void green_button_song();
}
```

Inheritance

```
class TeddyBear {
  public:
    TeddyBear(..);
    void roar_sound();
}
```

```
class RedTeddyBear {
  public:
    red_button_song();
}
```

```
class PurpleTeddyBear {
  public:
    purple_button_song();
}
```

```
class GreenTeddyBear {
  public:
    green_teddy_bear();
}
```

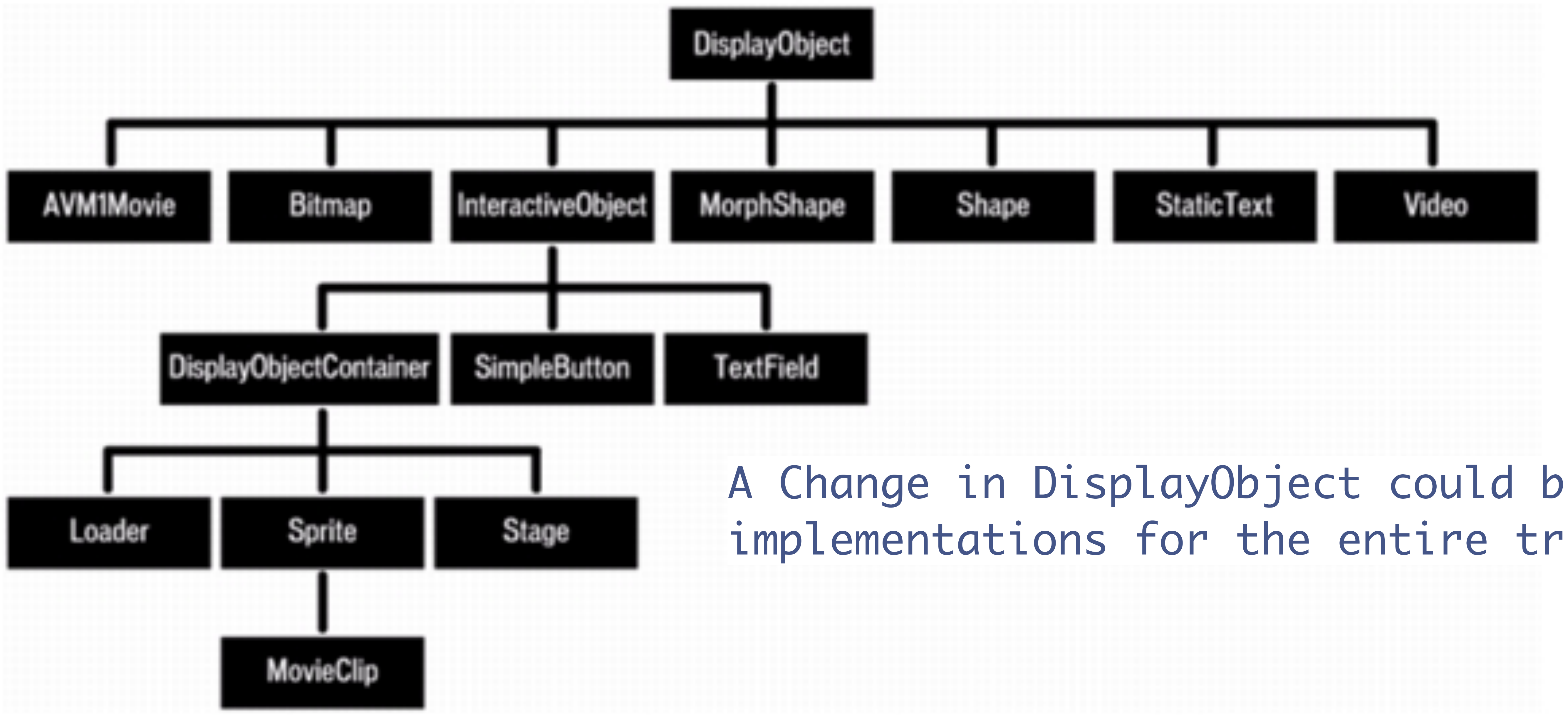
Lets take a look!

Inheritance

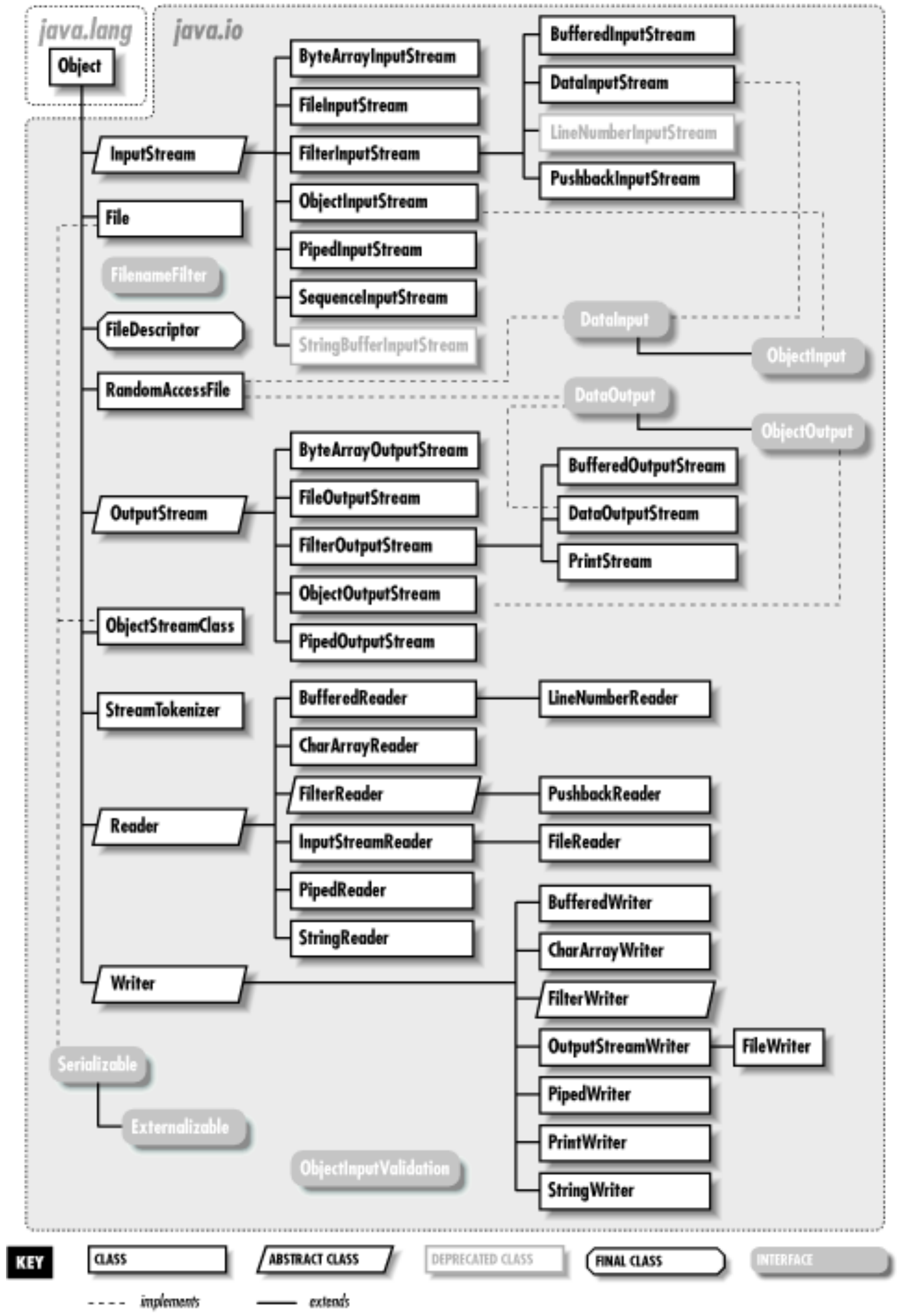
- With Inheritance, we were able to use the same implementation of one method across many different kinds of objects, brought together through a **parent-child** relationship.
- Child subclasses inherit **all** methods and attributes. (*constructors usually don't count here, depending on the language).* They can choose to override parent functions (green bear roaring differently)
- Big concept in languages like Java (where everything inherits one base **Object** class)

What might be the weaknesses of Inheritance?

Inheritance Trees



A Change in DisplayObject could break implementations for the entire tree!



Think about: maintaining and changing a large codebase over time

Aside: Two Other Keywords

- Object composition
  - Class A has instance variables of other class types.
  - Ex: want to produce multiple kinds of stuffed animals. Define things like “fur”, “feathers”, “claws”, “mouth”, etc., and compose them together into more complex stuffed animals.
  - Looser coupling; often a better choice than inheritance if possible
- Polymorphism
  - Different underlying types/implementations share a single interface
  - Ex: green bear inherits “roar” from (base) bear, but “roar” for green bear is implemented differently.

Traits

How else can we decompose?

https://play.rust-lang.org/?version=stable&mode=debug&edition=2018&gist=da8b2ac99e2c386656cb103c277a014e

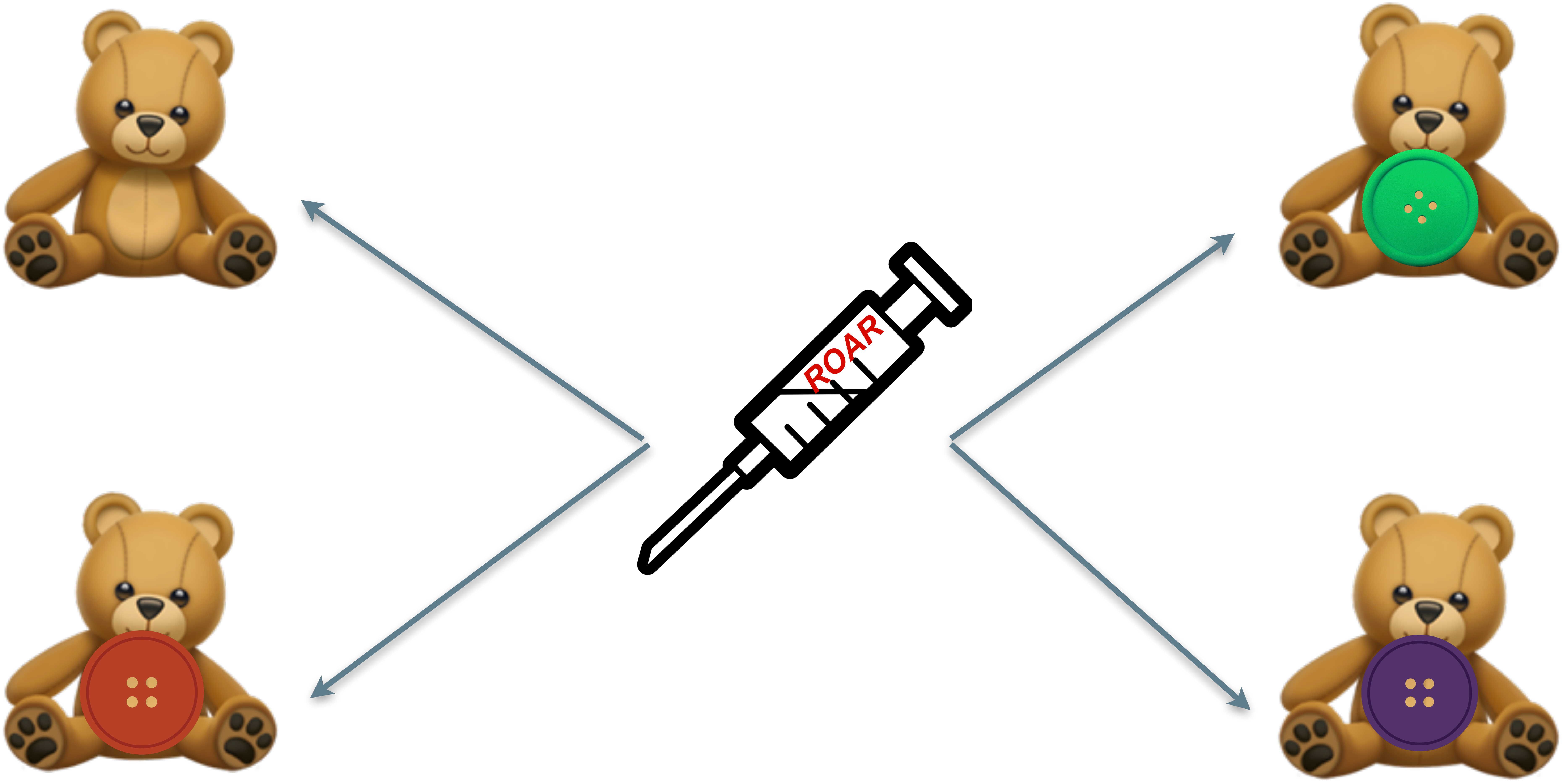
```
struct TeddyBear;

impl TeddyBear {
  fn roar(&self) {
    println!("ROAR!!");
  }
}
```

```
struct PurpleTeddyBear;

impl PurpleTeddyBear {
  fn roar(&self) {
    println!("ROAR!!");
  }
  fn purple_button_song(&self){
    /* Purple Song */
  }
}
```

```
struct RedTeddyBear;

impl RedTeddyBear {
  fn roar(&self) {
    println!("ROAR!!");
  }
  fn red_button_song(&self){
    /* Red Song */
  }
}
```

```
struct GreenTeddyBear;

impl GreenTeddyBear {
  fn roar(&self) {
    println!("ROAR!!");
  }
  fn green_button_song(&self){
    /* Green Song */
  }
}
```

Traits



Inject the code you want into the other classes! (Inject a trait into them!)

Let's make our first trait!

Traits Overview

- With traits, you write code that can be **injected** into any existing structure. (From TeddyBear to i32!) This code can have reference to **self,** so the code can be dependent on the instance
- Trait methods do not need to be fully defined - you could define a function that must be implemented when implementing a trait for a type. (Similar to Java interfaces)
- Traits can specify functions/data instances **should** have, instead of just getting many from another "parent".
- No more deep inheritance hierarchies. Just think: "Does this type implement this trait?"

Background, if you're interested:
https://blog.rust-lang.org/2015/05/11/traits.html

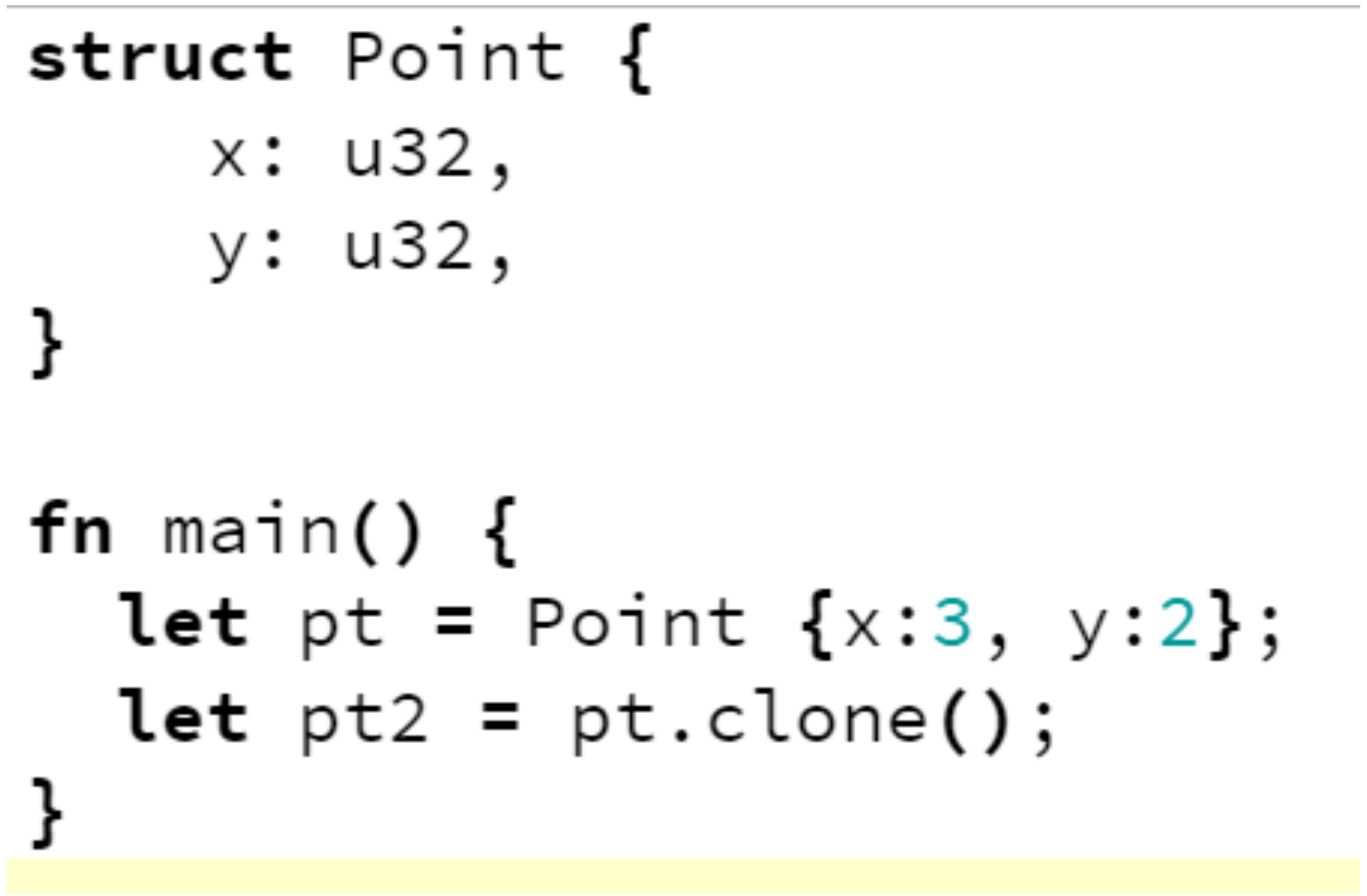
Questions?

Big Standard Rust Traits

Traits to Know

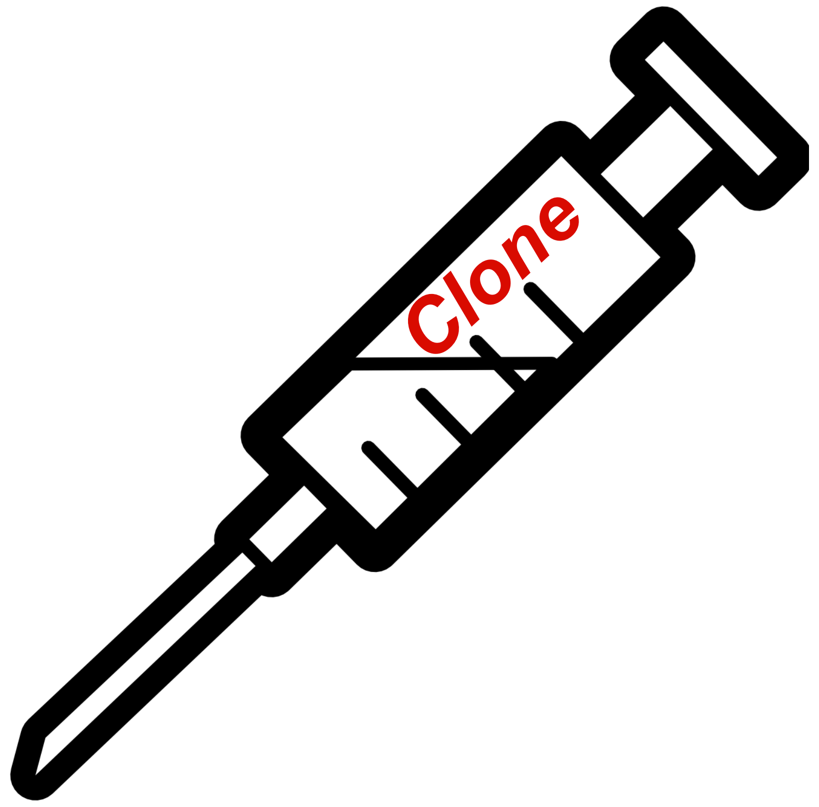
- **Copy:** Will create a new copy of an instance, instead of moving ownership when using assignment (=)
- **Clone:** Will return a new copy of an instance when calling the .clone() function on the method.
- **Drop:** Will define a way to free the memory of an instance - called when the instance reaches the end of the scope.
- **Display:** Defines a way to format a type, and show it (used by println!)
- **Debug:** Similar to Display, though not meant to be user facing (Meant for you to debug your types!)
- **Eq:** Defines a way to determine equality (defined by an equivalence relation) for two objects of the same type.
- **PartialOrd:** Defines a way to compare instances (less than, greater than, less than or equal to, etc.)

Lets implement a standard Trait!



```
struct Point {
    x: u32,
    y: u32,
}

fn main() {
  let pt = Point {x:3, y:2};
  let pt2 = pt.clone();
}
```

Does not compile - clone() isn't defined



Let's Inject Clone!

https://doc.rust-lang.org/std/clone/trait.Clone.html

Injecting Clone: recap

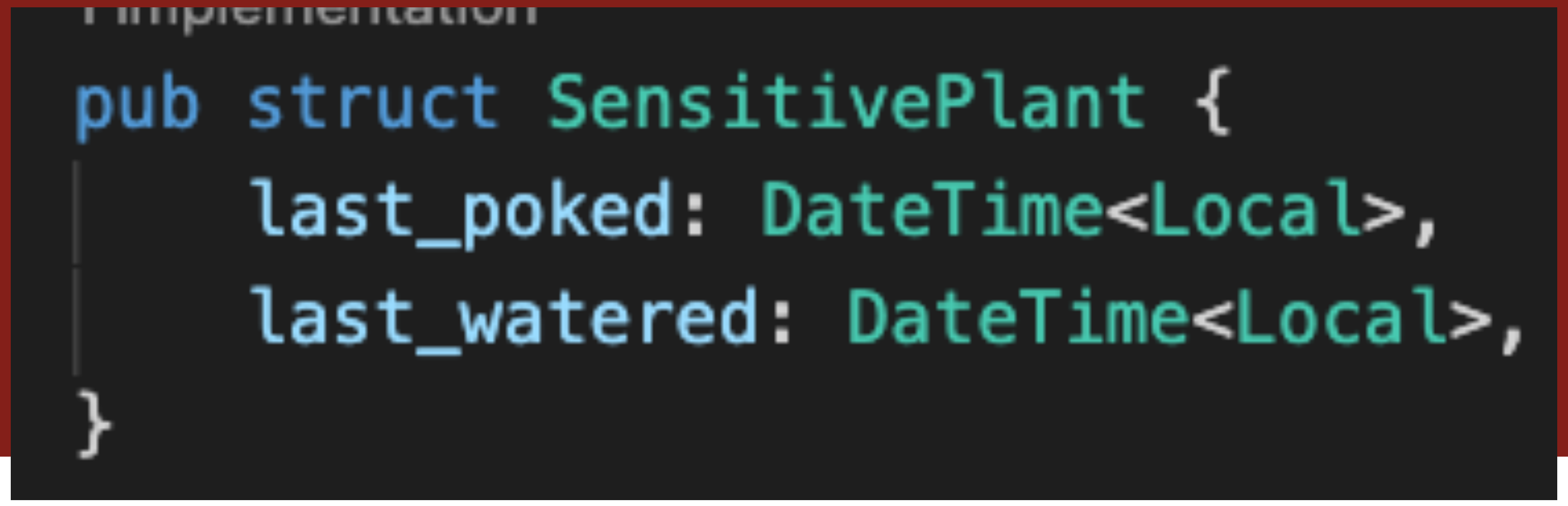
- You can implement any traits into any structure (as we did with Clone to Point), so long as they are compatible (**Drop** is not compatible with **Copy**)
- You can use the Rust Documentation as a way to tell you which functions need to be implemented, along with their parameter types.
- You can use **#[derive(x,y,z..)]** to *derive* traits. The Rust compiler will try to implement the traits for you, if your structure satisfies some rules (given by the documentation). IE: You can derive Clone if all members in the struct already implement Clone.

Next Time

- How can we write code that can accept many types?
- How can traits play a role in this?

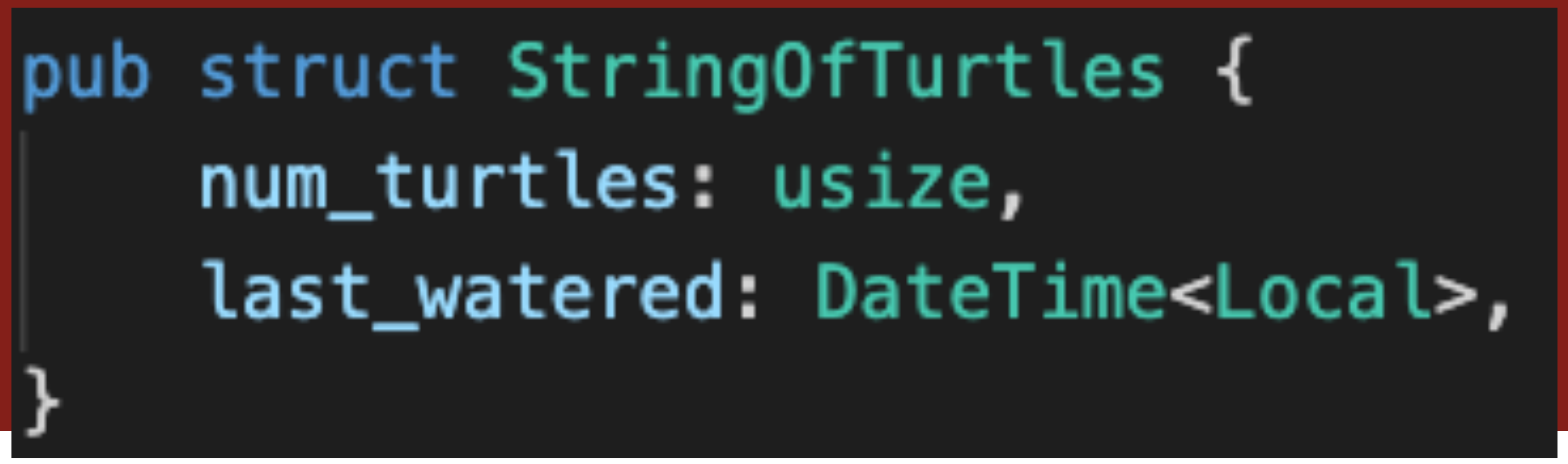
[Bonus slides] Week 5 exercises

- Idea: defining types that represent different kinds of plants.
- They'll all have custom implementations of traits like "water" and "needs_watering."
- We also will want to *derive* some helpful traits — e.g., printing out the current state of the plant for debugging purposes.

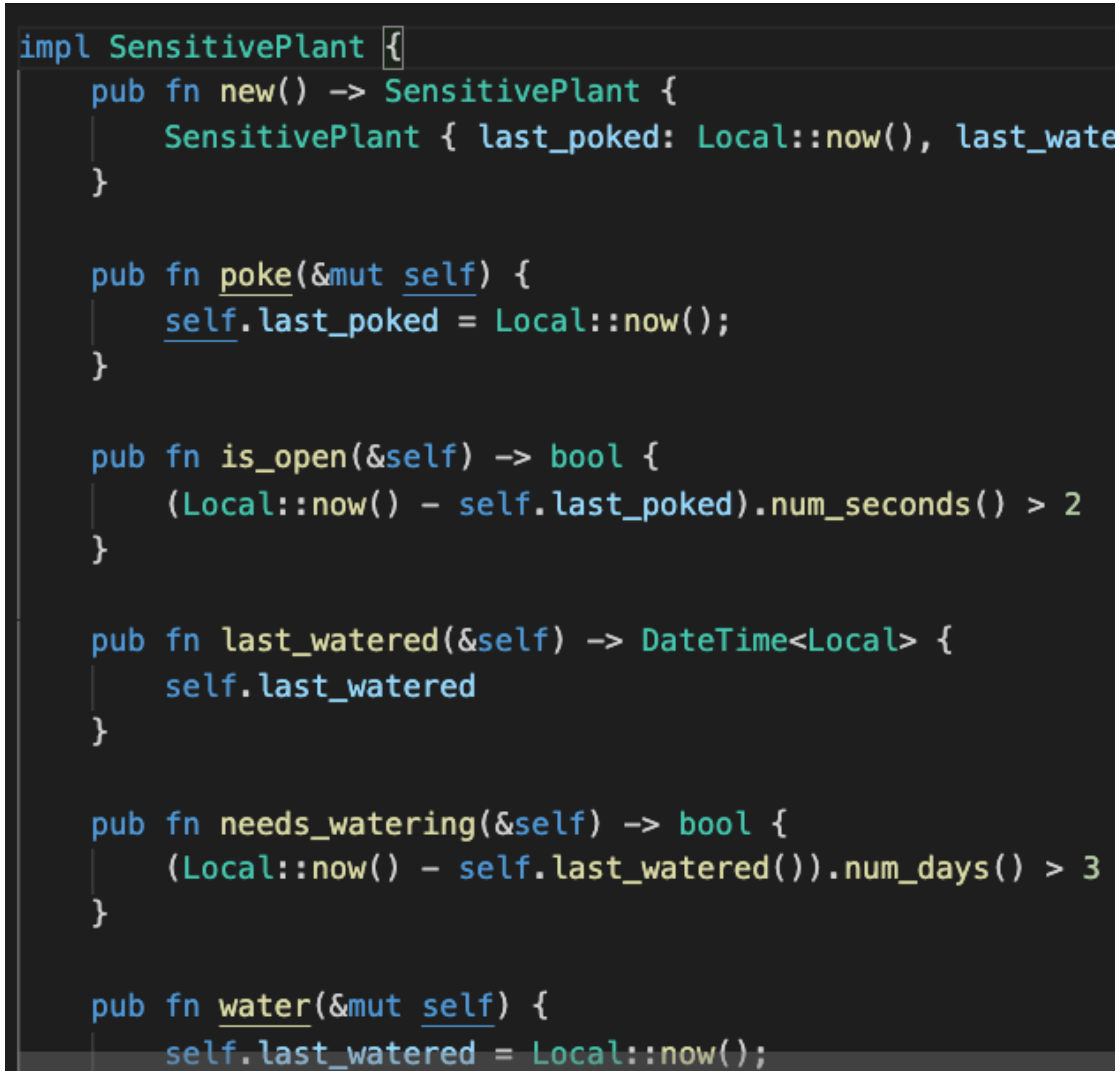


Implementation

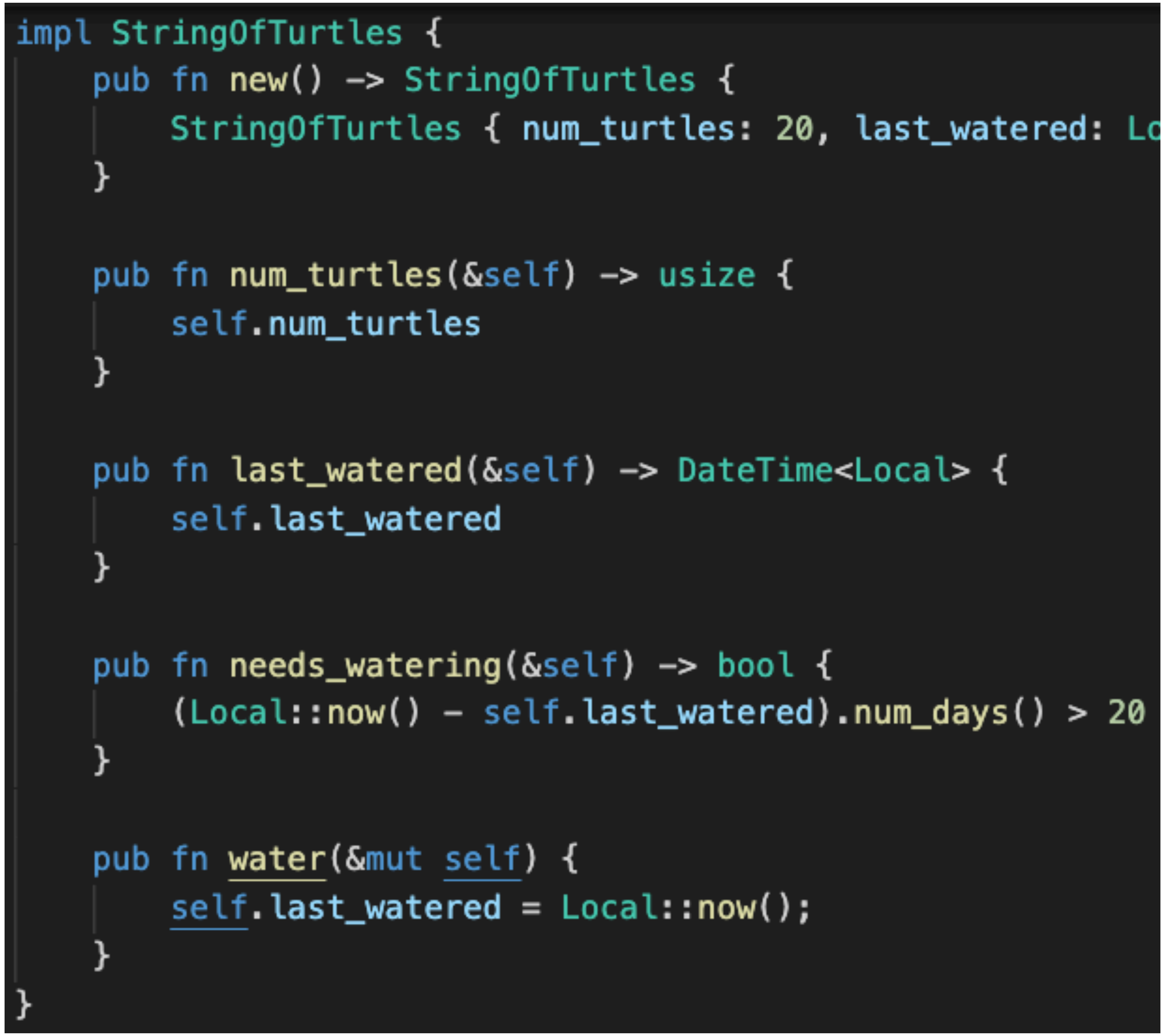
```rust
pub struct SensitivePlant {
    last_poked: DateTime<Local>,
    last_watered: DateTime<Local>,
}
```



```rust
pub struct StringOfTurtles {
    num_turtles: usize,
    last_watered: DateTime<Local>,
}
```



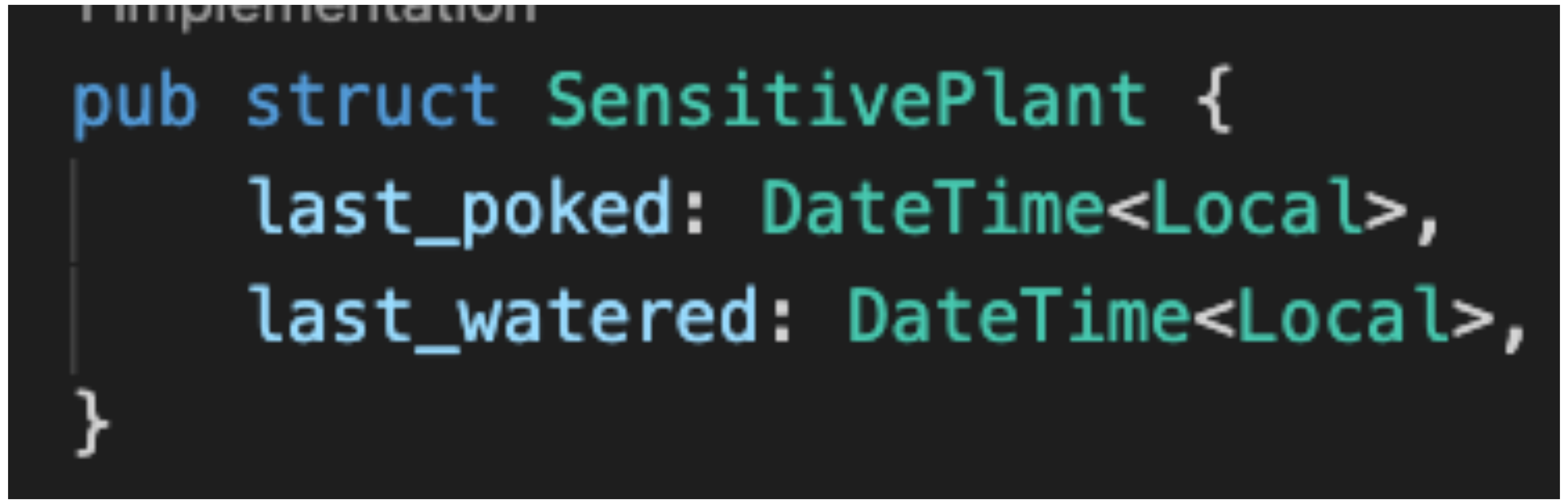
```rust
impl SensitivePlant {
    pub fn new() -> SensitivePlant {
        SensitivePlant { last_poked: Local::now(), last_wate
    }

    pub fn poke(&mut self) {
        self.last_poked = Local::now();
    }

    pub fn is_open(&self) -> bool {
        (Local::now() - self.last_poked).num_seconds() > 2
    }

    pub fn last_watered(&self) -> DateTime<Local> {
        self.last_watered
    }

    pub fn needs_watering(&self) -> bool {
        (Local::now() - self.last_watered()).num_days() > 3
    }

    pub fn water(&mut self) {
        self.last_watered = Local::now();
```



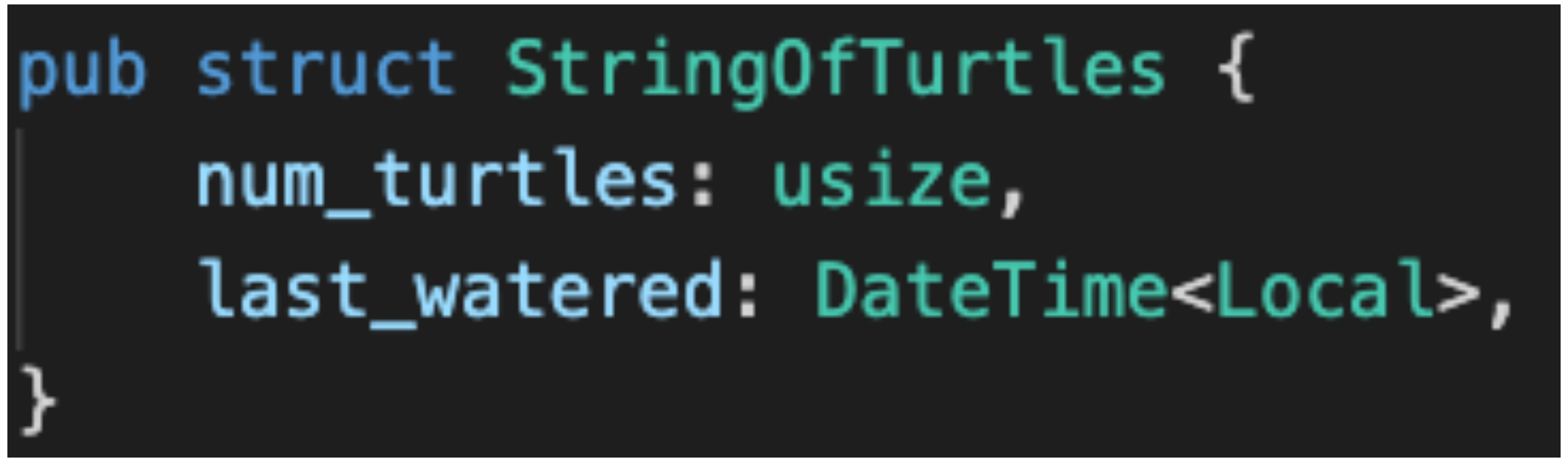
```rust
impl StringOfTurtles {
    pub fn new() -> StringOfTurtles {
        StringOfTurtles { num_turtles: 20, last_watered: Lo
    }

    pub fn num_turtles(&self) -> usize {
        self.num_turtles
    }

    pub fn last_watered(&self) -> DateTime<Local> {
        self.last_watered
    }

    pub fn needs_watering(&self) -> bool {
        (Local::now() - self.last_watered).num_days() > 20
    }

    pub fn water(&mut self) {
        self.last_watered = Local::now();
    }
}
```

Milestone: derive Debug

- Two common ways to print in Rust:
  - Display: clean, easy representation
    - Invoke with: **println!("{}", object);**
  - Debug: meant to be more verbose, for debugging
    - Invoke with: **println!(“{:?}”, object);**
- Can #derive Debug if all members are Debug
  - (note: `usize` and `DateTime` are Debug)
- **How would we do this here?**

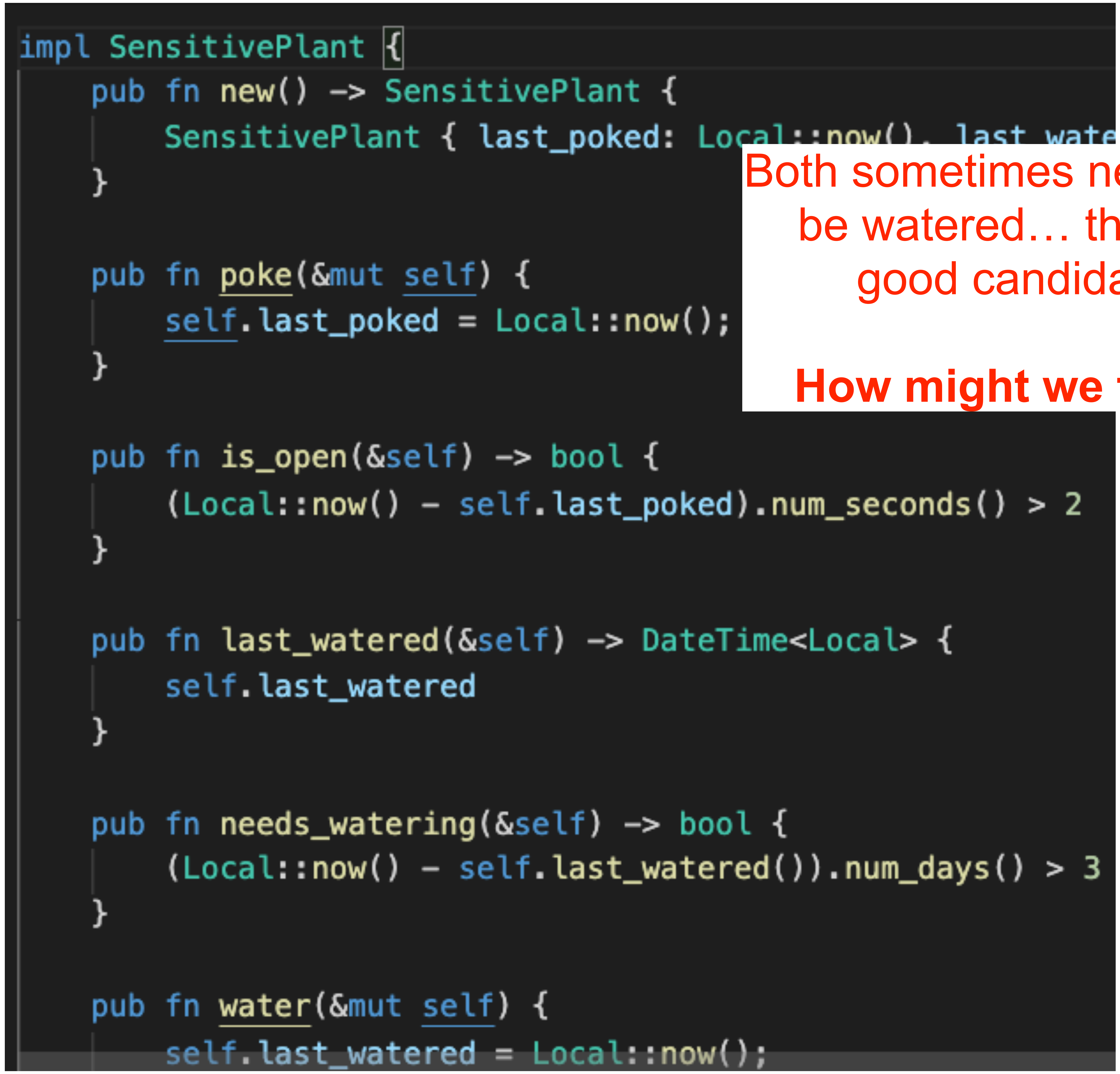


```
pub struct SensitivePlant {
    last_poked: DateTime<Local>,
    last_watered: DateTime<Local>,
}
```



```
pub struct StringOfTurtles {
    num_turtles: usize,
    last_watered: DateTime<Local>,
}
```

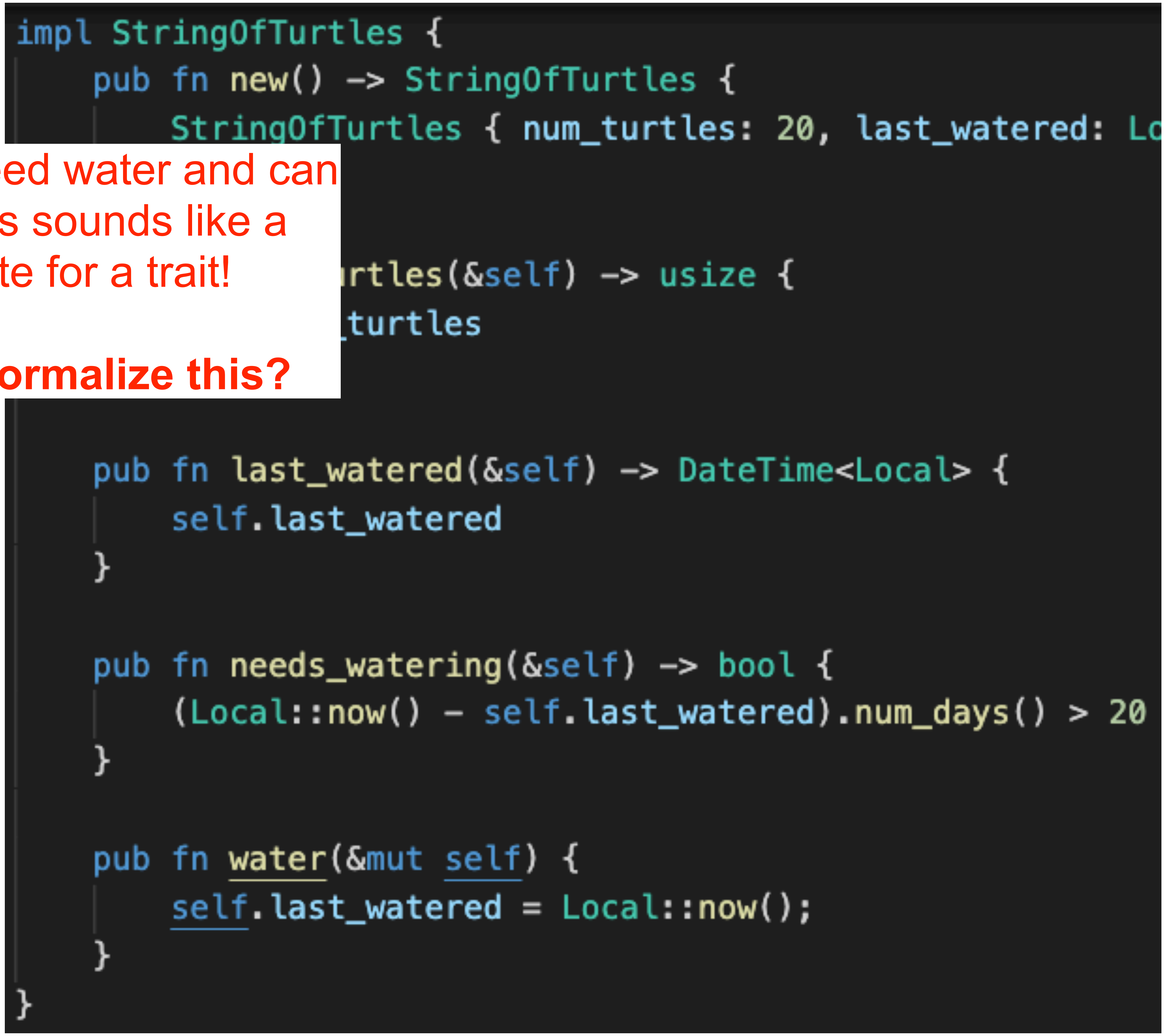
Milestone: functions -> traits



```
impl SensitivePlant {
    pub fn new() -> SensitivePlant {
        SensitivePlant { last_poked: Local::now(), last_wate
    }

    pub fn poke(&mut self) {
        self.last_poked = Local::now();
    }

    pub fn is_open(&self) -> bool {
        (Local::now() - self.last_poked).num_seconds() > 2
    }

    pub fn last_watered(&self) -> DateTime<Local> {
        self.last_watered
    }

    pub fn needs_watering(&self) -> bool {
        (Local::now() - self.last_watered()).num_days() > 3
    }

    pub fn water(&mut self) {
        self.last_watered = Local::now();
```

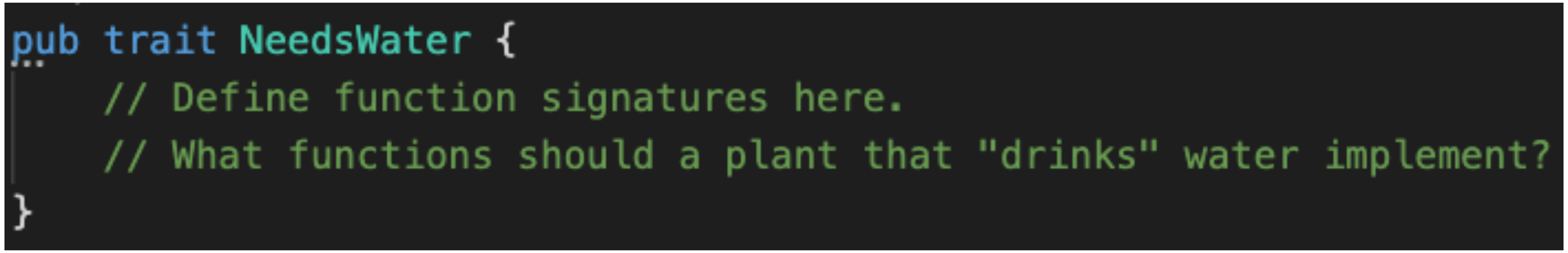
Both sometimes need water and can be watered… this sounds like a good candidate for a trait!

How might we formalize this?

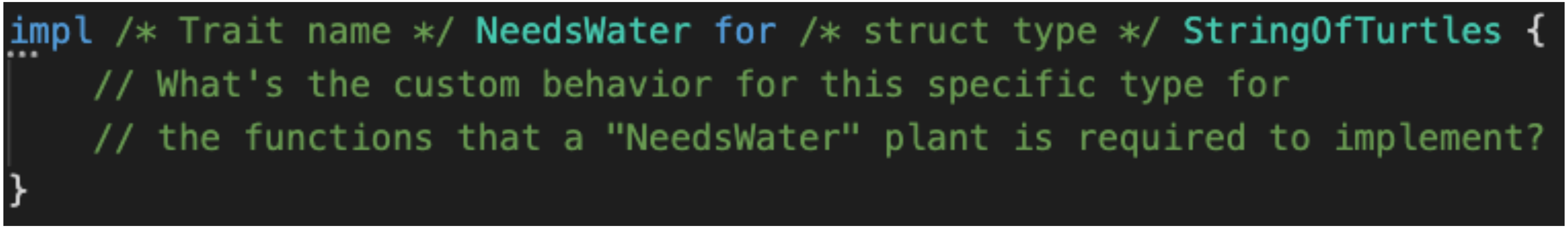


```
impl StringOfTurtles {
    pub fn new() -> StringOfTurtles {
        StringOfTurtles { num_turtles: 20, last_watered: Lo

                    rtles(&self) -> usize {
                    turtles


    pub fn last_watered(&self) -> DateTime<Local> {
        self.last_watered
    }

    pub fn needs_watering(&self) -> bool {
        (Local::now() - self.last_watered).num_days() > 20
    }

    pub fn water(&mut self) {
        self.last_watered = Local::now();
    }
}
```

Milestone: functions -> traits



```
pub trait NeedsWater {
    // Define function signatures here.
    // What functions should a plant that "drinks" water implement?
}
```

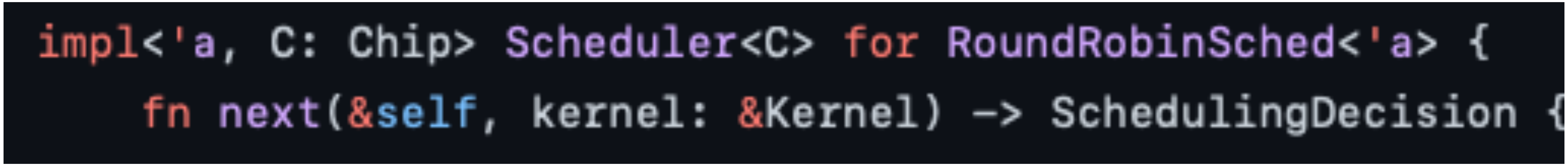


```
impl /* Trait name */ NeedsWater for /* struct type */ StringOfTurtles {
    // What's the custom behavior for this specific type for
    // the functions that a "NeedsWater" plant is required to implement?
}
```

Note: function signatures in `trait` def. must match function signatures in `impl` block. (I.e.: same names, same parameters.)

[Bonus slides] Traits IRL
Tock: Open-Source OS written in Rust

- In 110, you’ve learned about how an OS component called the scheduler “schedules” threads and processes (gives them time on CPU(s)).
- Multiple ways to implement this.
- Round robin = a popular scheduling implementation.
  - Run one thread/process for a time slice, then move on to the next one.
  - Like going around a circle of processes.



```
impl<'a, C: Chip> Scheduler<C> for RoundRobinSched<'a> {
    fn next(&self, kernel: &Kernel) -> SchedulingDecision {
```

https://github.com/tock/tock/

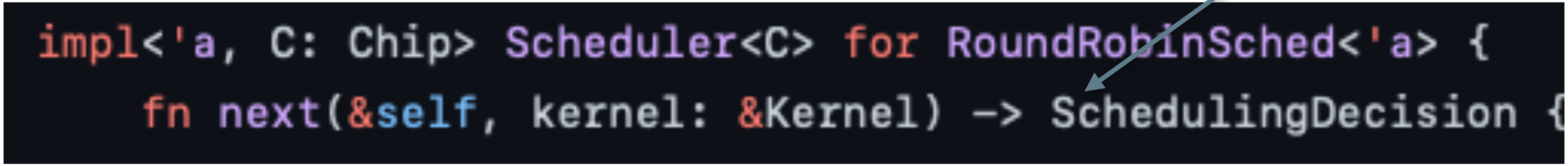
https://github.com/tock/tock/blob/master/kernel/src/scheduler/round_robin.rs

[Bonus slides] Traits IRL
Tock: Open-Source OS written in Rust

Generics (we’ll get to this next time)
What this means: this will work for multiple different *architectures* (pieces of hardware).

There’s a trait called “Scheduler”.
Multiple types implement “Scheduler.”
Here, we’re defining its specific implementation for the RR scheduler.

Example:
All schedulers must choose the NEXT process to run. Here’s the custom implementation for how the Round Robin scheduler does this!



```
impl<'a, C: Chip> Scheduler<C> for RoundRobinSched<'a> {
    fn next(&self, kernel: &Kernel) -> SchedulingDecision {
```

‘a indicates “lifetime”.
Out of scope for us, but if you’re interested, more here:
https://doc.rust-lang.org/rust-by-example/scope/lifetime.html

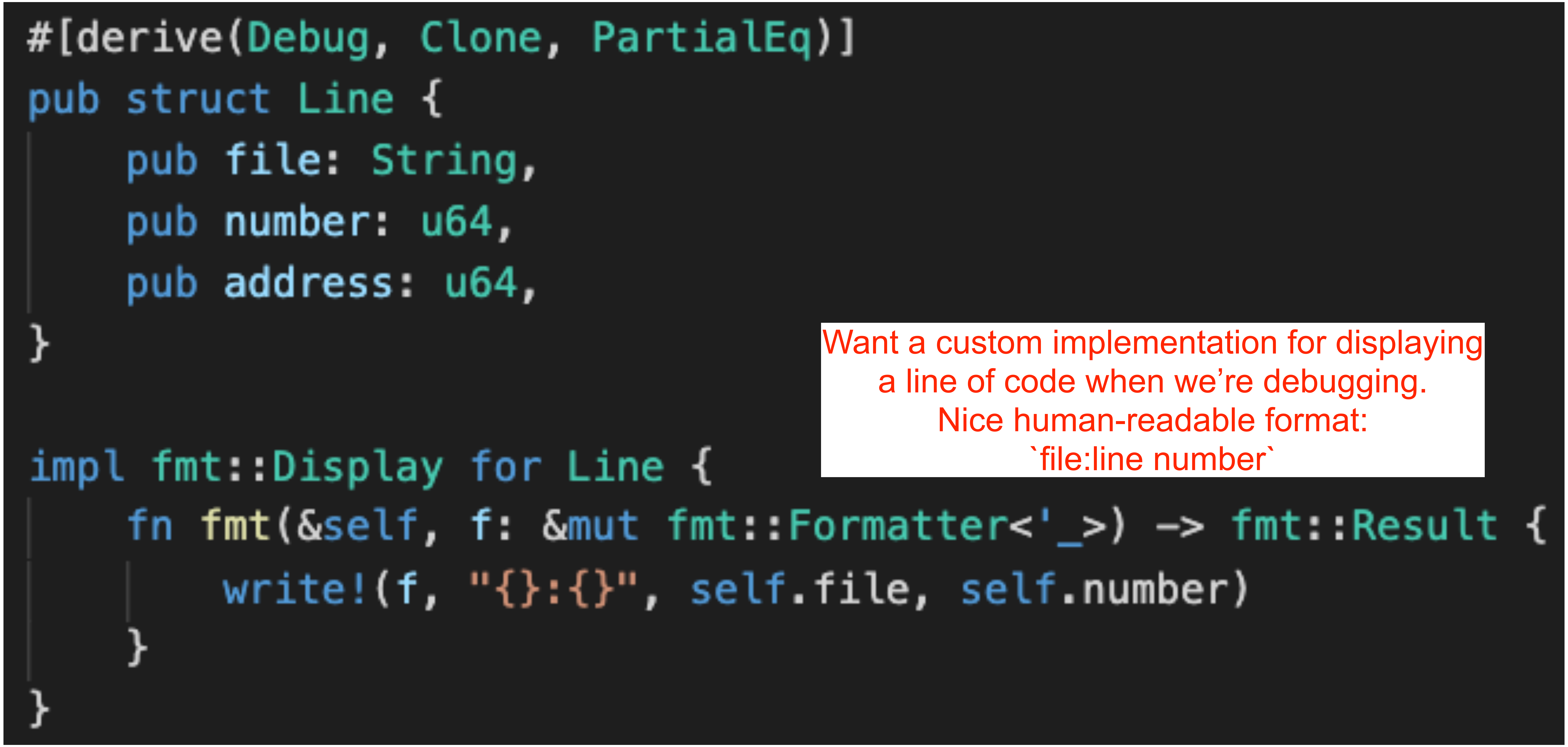
https://github.com/tock/tock/

https://github.com/tock/tock/blob/master/kernel/src/scheduler/round_robin.rs

[Bonus slides] Project 1 starter code examples

```
#[derive(Debug, Clone)]
2 implementations
pub enum Status {
    /// Indicates inferior stopped. Contains the signal that stopped the process, as well as the
    /// current instruction pointer that it is stopped at.
    Stopped(signal::Signal, u64),

    /// Indicates inferior exited normally. Contains the exit status code.
    Exited(i32),

    /// Indicates the inferior exited due to a signal. Contains the signal that killed the
    /// process.
    Signaled(signal::Signal),
}
```

[Bonus slides] Project 1 starter code examples



```
#[derive(Debug, Clone, PartialEq)]
pub struct Line {
    pub file: String,
    pub number: u64,
    pub address: u64,
}

impl fmt::Display for Line {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}:{}", self.file, self.number)
    }
}
```

Want a custom implementation for displaying a line of code when we're debugging. Nice human-readable format: `file:line number`