Make Your Own Stream Operators

Advanced stream processing in Rust

Willem Vanhulle

EuroRust 2025 • Paris, France

30 minutes + 10 minutes Q&A

Version with clickable links:
github.com/wvhulle/streams-eurorust-2025

Plan

Motivation 3
Rust's `Stream` trait 10
Consumption of streams 13
Example 1: Doubling integer streams (1-1) 16
Example 2: Cloning streams at run-time (1-N) 25
General principles 33
Bonus slides 37

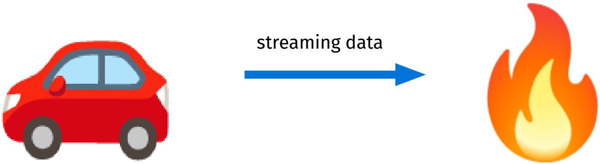
Motivation

My interest in stream processing (with Rust)

The problem: Processing incoming (streaming) data from moving vehicles

What I observed:

- *Inconsistent error handling* across the codebase
- Hard to reason about *data flow* and state
- Code duplication and boilerplate



`Streams` *in Rust are not new*

2019 — **async/await stabilized in Rust**
Stable async streams in std
RFC 2394, Rust 1.39.0

2009 — **Microsoft Reactive Extensions**
ReactiveX brings streams to mainstream
Erik Meijer, Microsoft

1997 — **Functional Reactive Programming**
Conal Elliott & Paul Hudak (Haskell)
ICFP '97, pp. 263-273

1978 — **Communicating Sequential Processes**
Tony Hoare formalizes concurrent dataflow
CACM 21(8):666-677

1973 — **Unix Pipes**
Douglas McIlroy creates `|` operator
Bell Labs, Unix v3-v4

1960s — **Dataflow Programming**
Hardware-level stream processing
Early dataflow architectures

What kind of streams exist?

Physical streams
Electronic signals
GPIO interrupts • UART frames • Network packets

OS abstraction

Leaf streams
OS/kernel constraints
tokio::fs::File • TcpListener • UnixStream • Interval

Requires an async runtime
('leaf future' by *Carl Fredrik Samson*)

Stream operators

Derived streams
Pure software transformations
map() • filter() • .double() • fork()

In this presentation

Naive stream processing

The challenge: Process TCP connections, filter messages, and collect 5 long ones

```rust
let mut filtered_messages = Vec::new(); let mut count = 0; let mut = 0;
let mut tcp_stream = tokio::net::TcpListener::bind("127.0.0.1:8080")
      .await?
      .incoming();
while let Some(connection) = tcp_stream.next().await {
    match connection {
        Ok(stream) => {
            if should_process(&stream) {
                // More nested logic needed...
            }
        }
        Err(e) => {
            total_errors += 1;
            log_connection_error(e);
            if total_errors > 3 { break; }
        } } }
```

Complexity grows with each requirement

Inside the processing block, **even more nested logic:**

```rust
// Inside the should_process(&stream) block:
match process_stream(stream).await {
    Ok(msg) if msg.len() > 10 => {
        filtered_messages.push(msg);
        count += 1;
        if count >= 5 { break; }  // Break from outer loop!
    }
    Ok(_) => continue,  // Skip short messages
    Err(e) => {
        total_errors += 1;
        log_error(e);
        if total_errors > 3 { break; }  // Another outer break!
    }
}
```

Problems: testing, coordination, control flow jumping around

Functional `Stream` *usage preview*

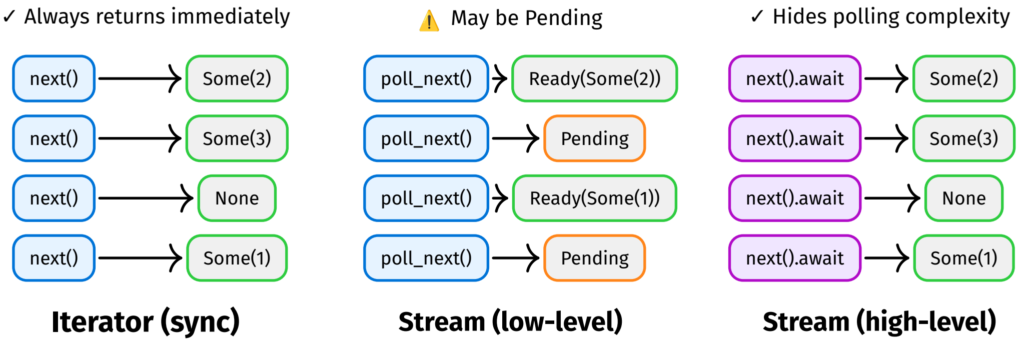
Same logic, much cleaner with stream operators:

```rust
let filtered_messages: Vec<String> = tcp_stream
    .filter_map(|connection| ready(connection.ok()))
    .filter(|stream| ready(should_process(stream)))
    .then(|stream| process_stream(stream))
    .filter_map(|result| ready(result.ok()))
    .filter(|msg| ready(msg.len() > 10))
    .take(5)
    .collect()
    .await;
```

"Programs must be written **for people to read**, and only incidentally for machines to execute." — *Harold Abelson & Gerald Jay Sussman*

Rust's `Stream` trait

***Moving from* `Iterator` *to* `Stream`**



A lazy interface

Similar to `Future`, but yields multiple items over time (when queried / **pulled**):

```rust
trait Stream {
    type Item;

    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context
    ) -> Poll<Option<Self::Item>>;
}
```

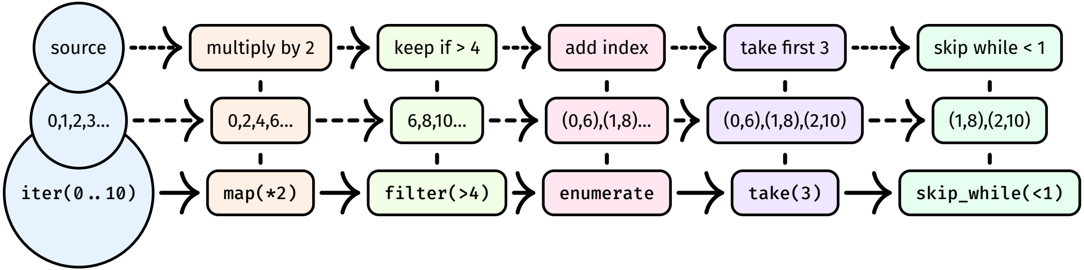
Returns `Poll` enum:

1. `Poll::Pending`: not ready (like `Future`)
2. `Poll::Ready(_)`:
   - `Ready(Some(item))`: new data is made available
   - `Ready(None)`: currently exhausted (not necessarily the end)

Consumption of streams

Building pipelines

The basic stream operators of `futures::StreamExt`:




```
stream::iter(0..10)
  .map(|x| x * 2)
  .filter(|&x| ready(x > 4))
  .enumerate().take(3).skip_while(|&(i, _)| i < 1)
```

The less-known `futures::ready` ***function***

Filter needs an **async closure** (or closure returning `Future`):

Option 1: Async block

```
stream.filter(|&x| async move {
  x % 2 == 0
})
```

Option 2: Async closure (Rust 2025+)

```
stream.filter(async |&x| x % 2 == 0)
```

Option 3: Wrap sync output with `std::future::ready()`

```
stream.filter(|&x| ready(x % 2 == 0))
```

`ready(value)` creates a `Future` that immediately resolves to `value`.

Bonus: `future::ready()` is `Unpin`, helping to keep the entire stream pipeline `Unpin` (if the input stream was `Unpin`)!

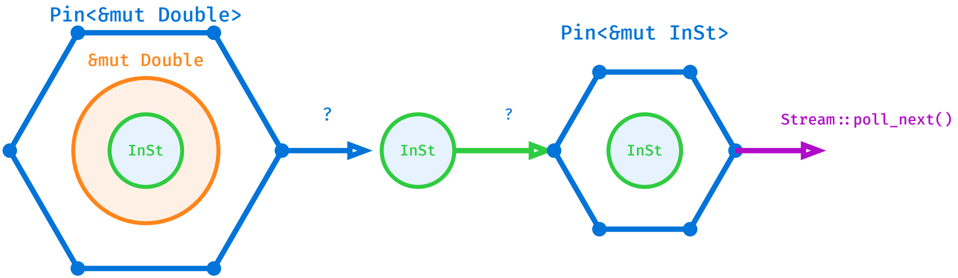
Example 1: Doubling integer streams (1-1)

Wrapping the original stream by value

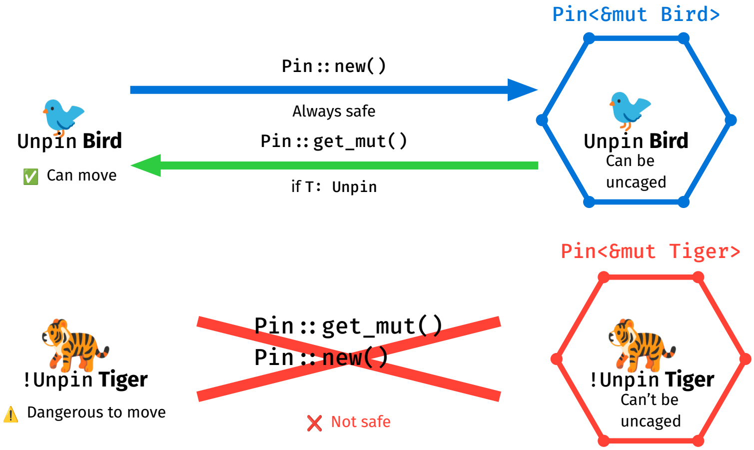
```rust
struct Double<InSt> { in_stream: InSt, }

impl<InSt> Stream for Double<InSt> where Stream: Stream<Item = i32> {
  type Item = InSt::Item;
  fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
      -> Poll<Option<Self::Item>> {
          Pin::new(self.in_stream) // ⚠ Will not compile!
              .poll_next(cx)
              .map(|x| x * 2)
  }
}
```

1. `Pin<&mut Self>` blocks access to `self.in_stream`!
2. `poll_next_unpin` requires `Unpin`

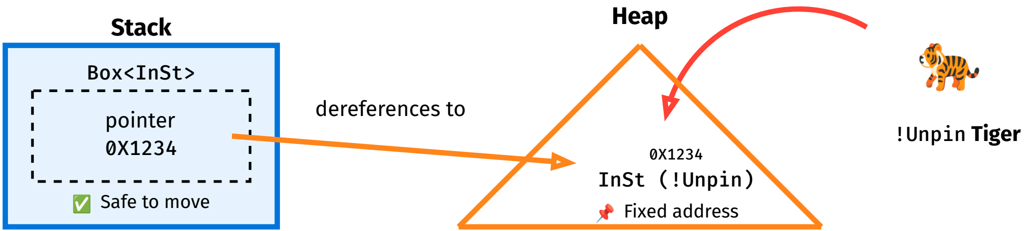
How to project to access `self.in_stream`***?***



`!Unpin` ***defends against unsafe moves***



Why **Box<T>: Unpin?**

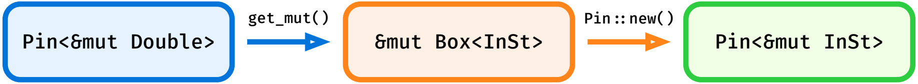


1. Put your **!Unpin** type on the heap with **Box::new()**
   (Heap content stays at fixed address)
2. The output of **Box::new(st)** is just a pointer
   (Moving pointers is safe)
3. Box<X>: Deref<Target = X>, so Box<InSt> **behaves like InSt**

```
struct Double {in_stream: Box<InSt>}: Unpin
```

Putting it all together visually

… and wrapping it around the boxed stream:



Complete `Stream` *trait implementation*

We can call `get_mut()` to get `&mut Double<InSt>` safely:

```
impl<InSt> Stream for Double<InSt>
where InSt: Stream<Item = i32>
{
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Option<Self::Item>>
    {
        // this: &mut Double<InSt>
        let this = self.get_mut(); // Safe because Double is Unpin
        match Pin::new(&mut this.in_stream).poll_next(cx) {
            Poll::Ready(r) => Poll::Ready(r.map(|x| x * 2)),
            Poll::Pending => Poll::Pending,
        }

    }
}
```

Distributing your operator

Create an extension trait to add `.double()` method to any integer stream:

```rust
trait DoubleStream: Stream {
    fn double(self) -> Double<Self>
    where Self: Sized + Stream<Item = i32>,
    { Double::new(self) }
}
```

Add a **blanket `impl`** that automatically implements `DoubleStream` for any `Stream<Item = i32>`:

```rust
impl<S> DoubleStream for S where S: Stream<Item = i32> {}
```

Important: A blanket implementation should be provided by you!

Users just add dependency + import

Super simple for users to adopt your custom operators:

```toml
[dependencies]
double-stream = "1.0"
```

The `DoubleStream` trait must be in scope to use `.double()`:

```rust
use double_stream::DoubleStream;  // Trait in scope

let doubled = stream::iter(1..=5).double();  // Now works!
```

Compositionality of traits (versus traditional OOP) shines!

Example 2: Cloning streams at run-time (1-N)

Problem: most streams aren't `Clone`

Latency may need to processed by different async tasks:

```
let tcp_stream = TcpStream::connect("127.0.0.1:8080").await?;
let latency = tcp_stream.latency(); // Stream<Item = Duration>
let latency_clone = latency.clone(); // Error! Can't clone stream
spawn(async move { process_for_alice(latency).await; });
spawn(async move { process_for_bob(latency_clone).await; });
```

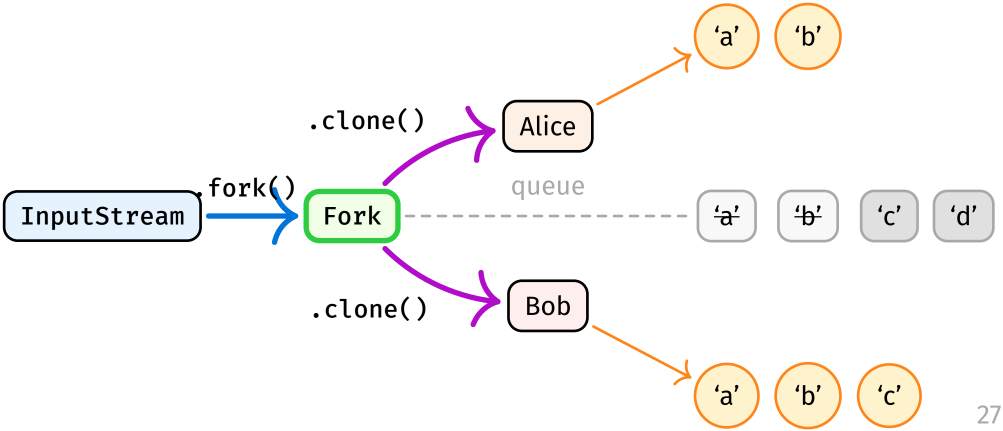
Solution: Create a ***stream operator*** that clones streams.

(Requirement: `Stream<Item: Clone>`, so we can clone the items)

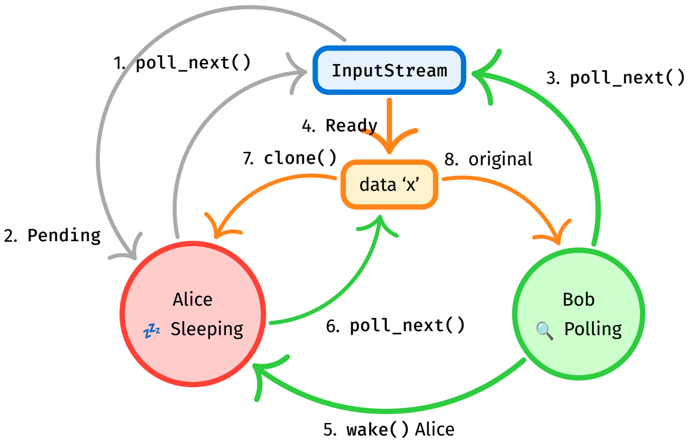
Approach:

1. Implement forking the input stream
2. Implement cloning on forked streams
3. Package as crate with blanket impl

Rough architecture of `clone-stream`



Polling and waking flow

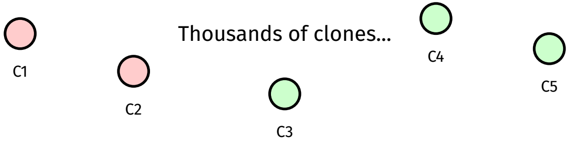


Complexity grows with thousands of clones

Careful state management:

Inherent async challenges:
- Dynamic clone lifecycle
- Memory leaks from orphaned wakers
- Cleanup when tasks abort
- Task coordination complexity

Stream-specific challenges:
- Ordering guarantees across clones
- Backpressure with slow consumers
- Sharing mutable state safely
- Avoiding duplicate items



Meaningful operator testing

When you build your own:

1. Pick an async run-time.
2. Define synchronization points with **`Barrier`**:

```
let b1 = Arc::new(Barrier::new(3)); //
First output
let b2 = b1.clone(); // Second output
let b3 = b1.clone(); // For input
```

3. Apply your custom operator

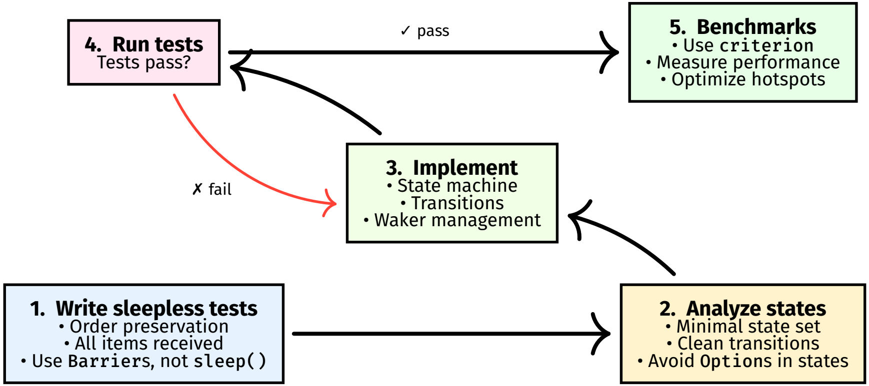
```
let stream1 = create_test_stream()
    .your_custom_operator();
let stream2 = stream1.clone();
```

Can be used for **benchmarks** too (use `criterion`).

Do not use `sleep(1ms)` in tests! (Use bariers!)

```
try_join_all([
    spawn(async move {
        setup_task().await;
        b1.wait().await;
        stream1.collect().await;
    }),
    spawn(async move {
        setup_task().await;
        b2.wait().await;
        stream2.collect().await;
    }),
    spawn(async move {
        b3.wait().await;
        send_to_stream().await;
    })
]).await.unwrap();
```

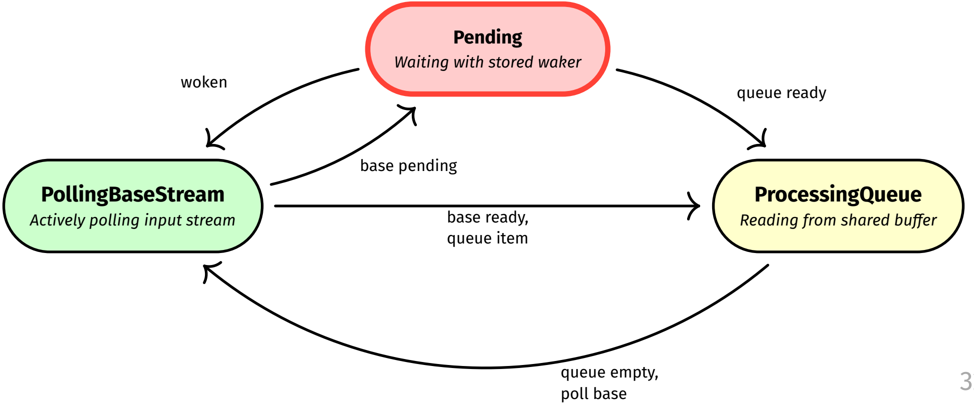
State machines for physically-separated components



“Perfection is achieved, not when there is nothing more to add, but when there is **nothing left to take away**.” — *Antoine de Saint-Exupéry*

State machine of `clone-stream`

Each clone maintains its own state:



General principles

Before building your own operators

1. For simple state machines: `futures::stream::unfold` constructor
2. Streams from scratch: `async-stream` crate with `yield`

Otherwise, import an operator from:

Standard: `futures::StreamExt`

- 5.7k ⭐, 342 contributors
- Since 2016, actively maintained
- Latest: v0.3.31 (Oct 2024)

RxJs-style: `futures-rx`

- Reactive operators & specialized cases
- 8 ⭐, small project
- Since Dec 2024, very new
- Fills gaps in `futures::StreamExt`

Build custom operators **only when no existing operator fits**!

Last recommendation

Don’t overuse streams:

- Keep pipelines short
- Only *physical async data flow*

Separation of concerns:

- Modular functions
- Descriptive names
- Split long functions

Use objective targets:

- Correctness unit tests
- Statistically relevant benchmarks

“When you have a hammer, everything looks like a nail.” — *Abraham Maslow*

Any questions?

Longer read: willemvanhulle.tech/blog/streams

Thank you!

Willem Vanhulle

Feel free to reach out!

willemvanhulle@protonmail.com
willemvanhulle.tech
github.com/wvhulle/streams-eurorust-2025

Bonus slides

The meaning of `Ready(None)`

Regular Stream

"No items **right now**"

(*Stream might yield more later*)

Fused Stream

"No items **ever again**"

(*Stream is permanently done*)

‘Fusing’ streams and futures

	Future	**Stream**	**Meaning**
Regular			May continue
Fused	**FusedFuture**	**FusedStream**	`is_terminated()` method
Fused			Done permanently
Fused value	Pending	Ready(None)	Final value

Flatten a finite collection of `Streams`

A finite collection of **`Streams`** = **`IntoIterator<Item: Stream>`**

```
let streams = vec![
    stream::iter(1..=3),
    stream::iter(4..=6),
    stream::iter(7..=9),
];

let merged = stream::select_all(streams);
```

1. Creates a **`FuturesUnordered`** of the streams
2. Polls all streams concurrently
3. Yields items as they arrive

Flattening an infinite stream

Beware!: `flatten()` on a stream of infinite streams will never complete!

```
let infinite_streams = stream::unfold(0, |id| async move {
    Some((stream::iter(id..), id + 1))
});
let flat = infinite_streams.flatten();
```

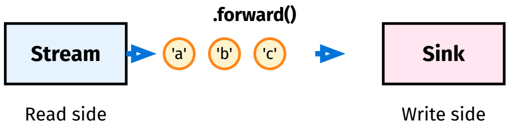
Instead, **buffer streams** concurrently with `flatten_unordered()`.

```
let requests = stream::unfold(0, |id| async move {
    Some((fetch_stream(format!("/api/data/{}", id)), id + 1))
});
let flat = requests.flatten_unordered(Some(10));
```

More `Stream` *features to explore*

Many more advanced topics await:

- **Boolean operations**: `any`, `all`
- **Async operations**: `then`
- **`Sinks`**: The write-side counterpart to `Streams`



The `Stream` trait: a lazy query interface

The `Stream` trait is NOT the stream itself - it's just a lazy frontend to query data.

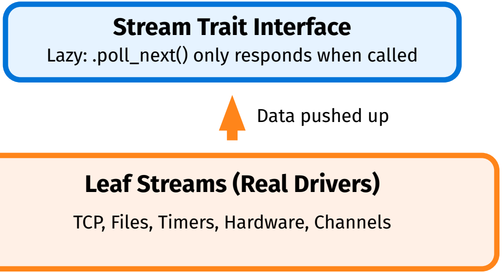
What `Stream` trait does:

- Provides uniform `.poll_next()` interface
- Lazy: only responds when asked
- Doesn't drive or produce data itself
- Just queries whatever backend exists

What actually drives streams:

- TCP connections receiving packets
- File I/O completing reads
- Timers firing
- Hardware signals
- Channel senders pushing data

The ‘real’ stream drivers



`Stream` trait just provides a **uniform way to query** - it doesn’t create or drive data flow.

Possible inconsistency

```rust
trait Stream {
    type Item;

    fn poll_next(self: Pin<&mut Self>, cx: &mut Context)
        -> Poll<Option<Self::Item>>
}
```

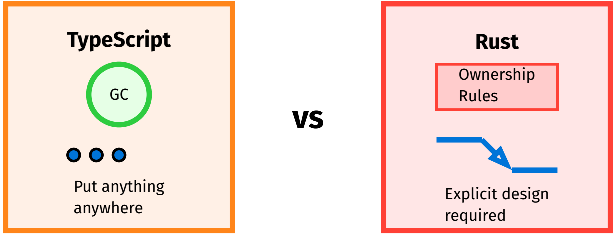
What about Rust rule **`self`** needs to be **`Deref<Target=Self>`**?

`Pin<&mut Self>` only implements **`Deref<Target=Self>`** for **`Self: Unpin`**.

Problem? No, **`Pin`** is an exception in the compiler.

Why does Rust bring to the table?

Reactivity in garbage collected languages is **completely different** from Rust’s ownership system



This fundamental difference explains why stream patterns from other languages don’t translate directly

The end