WRITE YOURSELF A CLI IN RUST

A **4-hour course** to learn hands-on Rust concepts for command-line tools

corrode

MEET THE TRAINER

Matthias Endler

- Rust consultant at corrode
- Started with Rust in 2015
- Hosted Hello Rust YouTube channel
- Hosts the Rust in Production podcast

corrode

ABOUT THE WORKSHOP

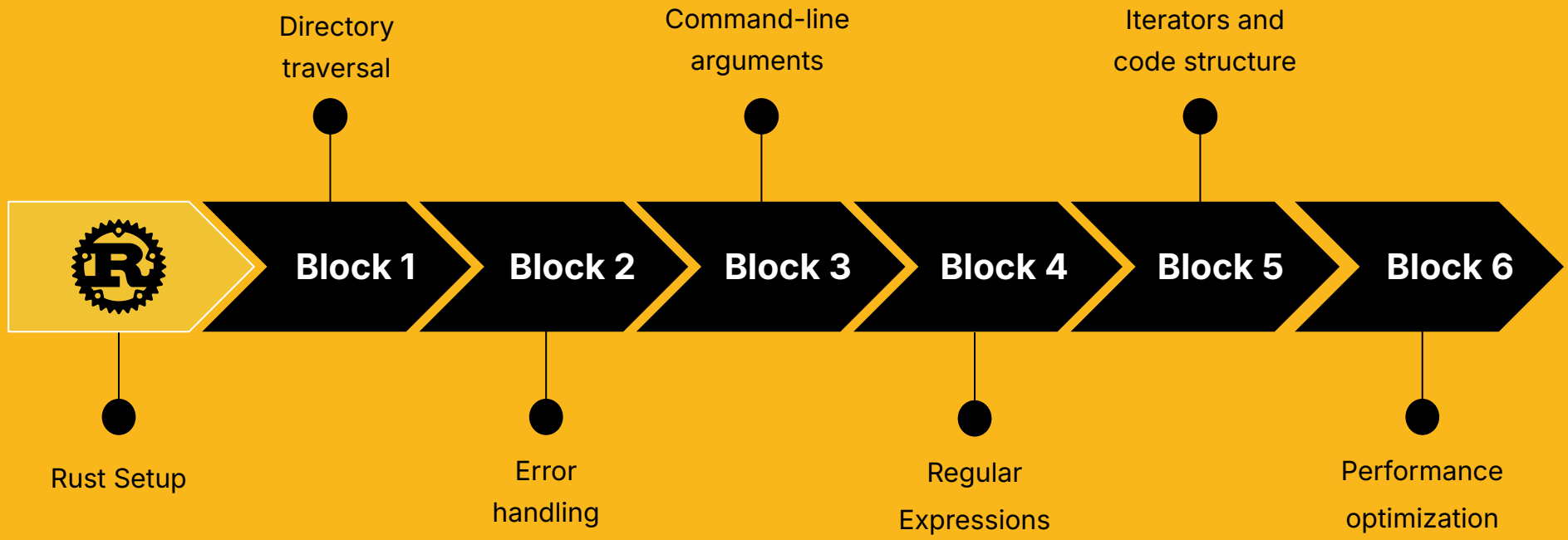
Goals

- Learn basic Rust concepts
- Work on a real-world project
- Use plain Rust; no dependencies
- Focus on idiomatic code

Structure

- 4 hours total
- Split up into six blocks
- Roughly 30min per block

SCHEDULE



corrode

BLOCK 0 - RUST SETUP

Main objective

- Install Rust using rustup or any other way.
- Run `rustc -V` to see if everything is okay.

Bonus track

- Set up rust analyzer for code completion (https://rust-analyzer.github.io/)
- Set up your project with additional clippy lints. (Example setup)

BLOCK 1 - EXECUTING A SINGLE SHELL COMMAND

Main objectives

- Write a command-line tool which recursively iterates over all files in a directory.
- Print the output to stdout.
- *Hint:* Take a look at fs::read_dir in the standard library to do that.

Bonus track

- Test some edge-cases (like invalid directories).
- Make the code as idiomatic as possible. (`cargo clippy` should run without errors.)
- Write a test to make sure the program works.

BLOCK 1 - PROJECT STRUCTURE

```
use std::{
     fs::{self},
     path::PathBuf,
     str::FromStr,
};

fn iter_files(path: PathBuf) {
  // Your code here
}

fn main() {
  iter_files(PathBuf::from_str(".").unwrap());
}
```

BLOCK 1 - REPOSITORY

https://github.com/corrode/write-yourself-a-cli

BLOCK 2 - ERROR HANDLING

Main objectives

- Introduce proper error handling.
- Don't use `unwrap` or `expect`
- Use the `?` operator to return errors.

Bonus track

- Introduce your own `Error` type
- `impl From<std::io::Error> for Error`
- Check out anyhow and thiserror.

BLOCK 3 - COMMAND-LINE ARGUMENTS

Main objectives

- Allow to pass a path to your CLI program, e.g.
  `cargo run -- /tmp`
- Only use the standard library for argument handling

Bonus track

- Handle multiple directories:
  `cargo run -- dir1 dir2`
- Handle flags:
  `cargo run -- --help`
- Check out clap.

BLOCK 4 - REGULAR EXPRESSIONS

Main objectives

- Allow filtering files using regular expressions:
  `cargo run -- dir pattern`
  e.g. to print all Rust files:
  `cargo run -- . '*.rs'`

Bonus track

- Write some tests for path matching
- Add glob support (hard)

BLOCK 5 - ITERATORS AND CODE STRUCTURE

Main objectives

- Implement `std::iter::Iterator` for your file finder

```
let finder = FileFinder::new(dir, pattern);

for path in finder.iter() {
  println!("{}", path.display());
}
```

Bonus track

- Add nice documentation
- More tests
- Can you print a tree-like structure?
- Compare `Iterator` with `futures::Stream`.

BLOCK 6 - PERFORMANCE OPTIMIZATION

Main objectives

- try it on a large directory like `/home`
- Benchmark your tool against `fd` and `fzf`
- Make some performance improvements.

Bonus track

- Use cargo-flamegraph to find bottlenecks.

BONUS - BRING YOUR OWN FEATURES!

Main objectives

- Everything is allowed. Some ideas:
  - Fix all clippy warnings
  - Add color to the output
  - Sort the search results by size, date, or name
  - Query the file system asynchronously
  - Add support for renaming files
  - Xargs mode (run commands for each match):

```
ff "\.json$" --chain "jq . {} | grep error"
```

SHOW AND TELL

WHAT WE LEARNED

- Basic Rust concepts
  - Standard library
  - File iteration
  - Argument parsing
- Idiomatic code
  - Iterator trait
  - Clippy
- Advanced topics
  - Regular expressions
  - Performance optimizations
  - crate ecosystem