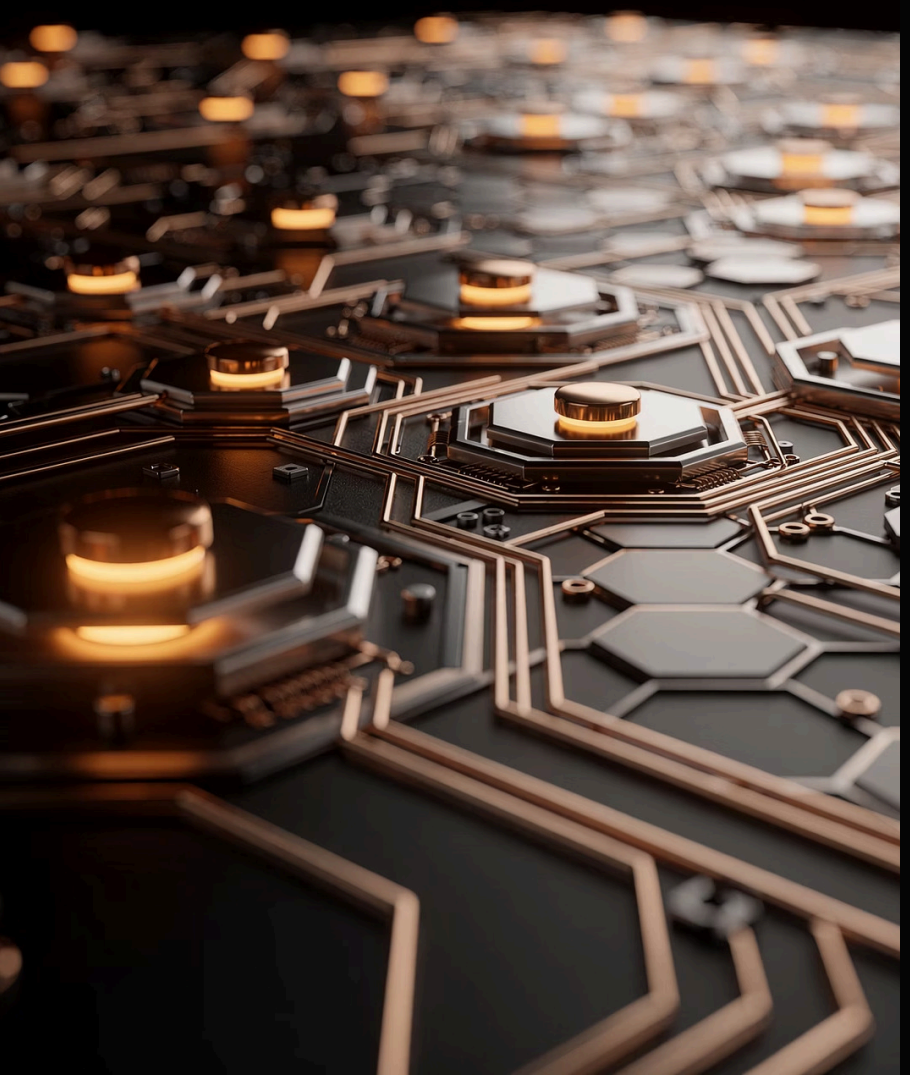


Rust Programming: Systems Programming Reimagined

Module 4: Traits, Lifetimes & Robust Error Handling

Presented by Chandrashekar Babu · training@chandrashekar.info

https://www.chandrashekar.info/ | https://www.slashprog.com/

Module 4 Overview

This module covers three pillars of idiomatic Rust programming — traits for polymorphism, lifetimes for memory safety, and error handling for robustness. Together they form the backbone of real-world Rust systems.

1

4.1 Traits

Shared behaviour, generics, operator overloading, dispatch

2

4.2 Lifetimes

Lifetime syntax, elision rules, borrow checker deep dive

3

4.3 Error Handling

panic!, Result<T,E>, ? operator, custom types

4

4.4 Lab

Config parser library with traits and full error handling

PART 4.1 — TRAITS

Defining Shared Behaviour

Traits are Rust's primary mechanism for defining shared behaviour across types — conceptually similar to interfaces in Java or Go, but more powerful and zero-cost at runtime.

What Are Traits?

A trait declares a set of method signatures that implementing types must provide. Traits can also supply **default implementations** that implementors may optionally override.

```
trait Summarizable {
    fn summary(&self) -> String;          // required

    fn author(&self) -> String {          // default implementation
        String::from("Unknown")
    }
}

struct NewsArticle { headline: String, location: String }

impl Summarizable for NewsArticle {
    fn summary(&self) -> String {
        format!("{} ({})", self.headline, self.location)
    }
    fn author(&self) -> String { String::from("Staff Writer") }
}

struct Tweet { username: String, content: String }

impl Summarizable for Tweet {
    fn summary(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
    // Uses default author() — no override needed
}
```

Both `NewsArticle` and `Tweet` satisfy the `Summarizable` contract while providing their own implementations — the essence of polymorphism in Rust.

Trait Syntax Deep Dive

Trait Members

Traits can declare **required methods** (no body), **provided methods** (with default body), **associated types**, and **associated constants**.

```
trait MyTrait {
    fn required(&self) -> i32;
    fn provided(&self) -> i32 {
        self.required() * 2
    }
    type Output;
    const ID: u32;
}
```

Multiple Traits per Type

A single struct can implement any number of traits, enabling rich composition without class hierarchies.

```
trait Shape {
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
    fn name(&self) -> &'static str;
}

struct MyStruct;
impl TraitA for MyStruct { fn a(&self) { } }
impl TraitB for MyStruct { fn b(&self) { } }
impl Shape for MyStruct { ... }
```

There is no limit on how many traits a type may implement, making composition highly expressive.

The Orphan Rule & Coherence

Rust enforces the **orphan rule** to prevent conflicting implementations across crates: you may implement a trait for a type only if *at least one* of the two is defined in your crate.

✅ Allowed

Implement `std::fmt::Display` for your own `MyType`. The type is yours.

Implement your own `MyTrait` for `i32`. The trait is yours.

❌ Not Allowed

Implement `std::fmt::Display` for `i32`. Both are from the standard library — outside your crate.

This rule guarantees **trait coherence**: there is always exactly one implementation for a given (trait, type) pair, preventing ambiguity at link time and making Rust's type system decidable.

Trait Bounds on Generic Functions

Rust provides several syntactic forms for expressing trait requirements on generic parameters. All are equivalent — choose the one that reads most clearly for your use case.

```
// 1. impl Trait syntax — concise for simple cases
fn notify(item: &impl Summarizable) {
 println!("Breaking: {}", item.summary());
}

// 2. Trait bound syntax — more explicit
fn notify<T: Summarizable>(item: &T) { ... }

// 3. Multiple bounds — combine with +
fn notify_debug<T: Summarizable + std::fmt::Debug>(item: &T) { ... }

// 4. where clause — clearest for complex bounds
fn complex_function<T, U>(t: &T, u: &U) -> String
where
 T: Summarizable + Clone,
 U: Summarizable + std::fmt::Debug,
{ ... }

// 5. Return position impl Trait
fn returns_summarizable() -> impl Summarizable {
 Tweet { username: "rustlang".into(), content: "Traits!".into() }
}
```

Conditional Trait Implementations

Rust lets you implement methods *only when* a generic parameter satisfies certain traits — a technique that enables **zero-cost abstractions**.

```
struct Pair<T> { x: T, y: T }

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self { Self { x, y } }
}

// cmp_display available ONLY when T: Display + PartialOrd
impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y { println!("largest: x = {}", self.x); }
        else { println!("largest: y = {}", self.y); }
    }
}

// Blanket impl — every type that is Display becomes Print for free
trait Print { fn print(&self); }
impl<T: Display> Print for T {
    fn print(&self) { println!("Value: {}", self); }
}
```

Key insight: Blanket implementations let you extend behaviour across the entire type ecosystem without modifying individual types — the foundation of how `ToString` is implemented for everything that implements `Display`.

Operator Overloading via Traits

Rust exposes operator overloading through the `std::ops` trait family. Each arithmetic or bitwise operator maps to a corresponding trait that you can implement for your own types.

```
use std::ops;

#[derive(Debug, Clone, Copy)]
struct Complex { real: f64, imag: f64 }

impl ops::Add for Complex {
    type Output = Complex;
    fn add(self, o: Complex) -> Complex {
        Complex { real: self.real + o.real,
                  imag: self.imag + o.imag }
    }
}

impl ops::Mul for Complex {
    type Output = Complex;
    fn mul(self, o: Complex) -> Complex {
        Complex {
            real: self.real*o.real - self.imag*o.imag,
            imag: self.real*o.imag + self.imag*o.real,
        }
    }
}

impl ops::AddAssign for Complex {
    fn add_assign(&mut self, o: Complex) {
        *self = *self + o;
    }
}
```

Overridable Operators

Arithmetic

Add, Sub, Mul, Div, Rem, Neg

Bitwise

BitAnd, BitOr, BitXor, Shl, Shr, Not

Indexing

Index, IndexMut

Deref

Deref, DerefMut

Callable

Fn, FnMut, FnOnce

Custom Indexing with `ops::Index`

Beyond arithmetic, you can make your types indexable with arbitrary key types — useful for matrix, graph, or grid abstractions.

```
struct Matrix { data: [[f64; 3]; 3] }

impl ops::Index<(usize, usize)> for Matrix {
    type Output = f64;
    fn index(&self, (row, col): (usize, usize)) -> &f64 {
        &self.data[row][col]
    }
}

fn main() {
    let m = Matrix { data: [[1.0,2.0,3.0],[4.0,5.0,6.0],[7.0,8.0,9.0]] };
    println!("{}", m[(1, 1)]); // 5.0 — tuple indexing!
}
```

type Output

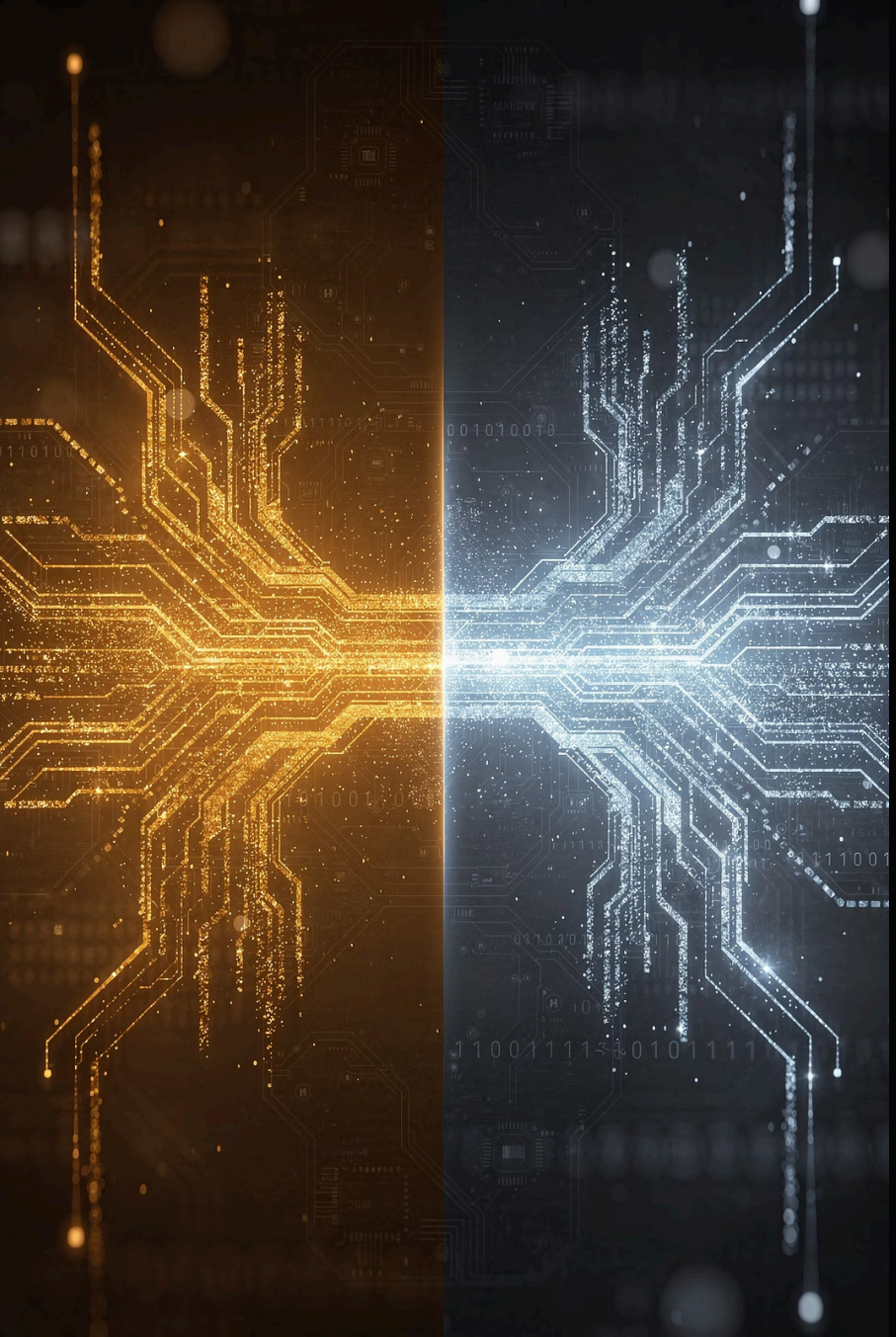
Specifies what the index operation returns — can differ from the element type.

Idx Type

Any type can serve as the index key — tuples, enums, custom structs.

IndexMut

Implement `IndexMut` separately to support `m[(r,c)] = value`.



Trait Objects & Dynamic Dispatch

Rust supports two dispatch strategies for polymorphic code. The choice between them is a deliberate trade-off between binary size, runtime overhead, and flexibility.

Static vs Dynamic Dispatch Explained

Static Dispatch — Monomorphisation

```
// Compiler generates a separate function
// for each concrete type T
fn draw_static<T: Draw>(item: T) {
    item.draw(); // inlined!
}
draw_static(Circle); // → draw_static_Circle
draw_static(Square); // → draw_static_Square
```

- ✅ Zero runtime overhead
- ✅ Inlining and optimisation possible
- ❌ Binary size grows with each concrete type

Dynamic Dispatch — Trait Objects

```
fn draw_dynamic(item: &dyn Draw) {
    item.draw(); // vtable lookup at runtime
}

// Heterogeneous collections — only possible
// with trait objects!
let shapes: Vec<&dyn Draw> = vec![&Circle, &Square];
for shape in shapes { shape.draw(); }
```

- ✅ Single compiled function
- ✅ Enables heterogeneous collections
- ❌ Small vtable lookup overhead
- ❌ No inlining across dispatch boundary

The memory layout of `&dyn Draw` is a **fat pointer**: a data pointer to the concrete value, and a vtable pointer containing `drop`, `size`, `align`, and pointers to each method implementation.

Object Safety Rules

Not every trait can be used as a trait object (`&dyn Trait`). A trait must be **object-safe** — the compiler enforces two key rules.

Rule 1 — No `Self` return

```
//
```

Supertraits — Trait Inheritance

A trait can declare that implementors must *also* implement another trait — called a **supertrait**. This enables method reuse and expressive type hierarchies without class inheritance.

```
trait Shape { fn area(&self) -> f64; }

trait Circle: Shape { // Shape is a supertrait of Circle
 fn radius(&self) -> f64;
 fn diameter(&self) -> f64 { self.radius() * 2.0 }
 fn circumference(&self) -> f64 {
 2.0 * std::f64::consts::PI * self.radius()
 }
}

struct MyCircle { r: f64 }

impl Shape for MyCircle {
 fn area(&self) -> f64 { std::f64::consts::PI * self.r * self.r }
}
impl Circle for MyCircle {
 fn radius(&self) -> f64 { self.r }
}

fn print_circle_info(c: &impl Circle) {
 println!("Area: {:.2}, Circumference: {:.2}",
 c.area(), // from Shape
 c.circumference() // from Circle
 );
}
```

Associated Types in Traits

Associated types are **placeholder types** declared inside a trait that the implementor must specify. Unlike generic parameters, they allow only one implementation per type — making APIs cleaner and more readable.

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}

struct Counter { count: u32 }

impl Iterator for Counter {
    type Item = u32;
    fn next(&mut self) -> Option<u32> {
        if self.count < 5 {
            self.count += 1;
            Some(self.count)
        } else { None }
    }
}

fn sum_iterator<I: Iterator<Item=u32>>(iter: &mut I) -> u32 {
    let mut total = 0;
    while let Some(v) = iter.next() { total += v; }
    total
}
```

Generics vs Associated Types

Generics allow multiple implementations per type: `impl Converter<String> for MyType` and `impl Converter<i32> for MyType` can coexist.

Associated types allow exactly one: once you write `type Item = u32` inside an `impl Iterator` block, the relationship is fixed for that type.

Use associated types when a type has a natural, singular output type — iterators, futures, and smart pointers are canonical examples.

PART 4.2 — LIFETIMES

Lifetimes & Compiler Deep Dive

Lifetimes are Rust's mechanism for ensuring that every reference always points to valid memory — checked entirely at compile time, with zero runtime cost.

What Are Lifetimes?

Every reference in Rust has a **lifetime** — the scope during which that reference must remain valid. Most of the time, the compiler infers lifetimes automatically. Annotations become necessary when the compiler cannot determine which lifetime a returned reference belongs to.

```
// Implicit lifetime — compiler infers it
fn first_word(s: &str) -> &str {
 let bytes = s.as_bytes();
 for (i, &item) in bytes.iter().enumerate() {
 if item == b' ' { return &s[0..i]; }
 }
 &s[..]
}

// The explicit version (what the compiler sees internally)
fn first_word<'a>(s: &'a str) -> &'a str { ... }

// Why lifetimes prevent dangling references:
fn main() {
 let result;
 {
 let s = String::from("hello");
 result = first_word(&s); // borrows from s
 } // s dropped here!
 // println!("{}", result); // ERROR — dangling reference
}
```

Lifetime Syntax

Lifetime annotations use the syntax `'a`, `'b`, `'static`, etc. They describe *relationships* between lifetimes — they do not change how long anything lives.

```
// Single lifetime — x and y must live at least as long as the return value
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
 if x.len() > y.len() { x } else { y }
}

// Multiple independent lifetimes
fn complex<'a, 'b>(x: &'a str, _y: &'b str) -> &'a str { x }

// Lifetime in structs — struct cannot outlive the reference it holds
struct ImportantExcerpt<'a> { part: &'a str }

impl<'a> ImportantExcerpt<'a> {
 fn level(&self) -> i32 { 3 }
 fn announce_and_return_part(&self, announcement: &str) -> &str {
 println!("Attention: {}", announcement);
 self.part // lifetime elided — returns &'a str (Rule 3)
 }
}

fn main() {
 let novel = String::from("Call me Ishmael...");
 let first_sentence = novel.split('.').next().unwrap();
 let excerpt = ImportantExcerpt { part: first_sentence };
}
```

Lifetime Elision Rules

Rather than requiring annotations everywhere, the compiler applies three **elision rules** that cover the vast majority of real-world functions. If the rules fully resolve all output lifetimes, no annotation is needed.

1 Each input reference gets its own lifetime

`fn compare(s1: &str, s2: &str)` → `fn compare<'a, 'b>(s1: &'a str, s2: &'b str)`

2 If exactly one input lifetime, it propagates to all outputs

`fn first_word(s: &str) -> &str` → `fn first_word<'a>(s: &'a str) -> &'a str`

3 If one of the inputs is `&self` or `&mut self`, its lifetime goes to all outputs

Applies to methods — the receiver dominates. No annotation required in the common case.

Lifetime Annotations in Structs

When a struct holds a reference, the struct itself becomes parameterised by a lifetime, expressing the constraint that the struct cannot outlive the data it references.

```
struct RefHolder<'a> {
 reference: &'a i32, // struct tied to the lifetime of this reference
 count: i32, // owned fields need no annotation
}

struct MultiRef<'a, 'b> { first: &'a str, second: &'b str }

impl<'a> RefHolder<'a> {
 fn get_ref(&self) -> &i32 { self.reference } // Rule 3 elides lifetime
 fn compare<'b>(&self, other: &'b i32) -> bool {
 *self.reference == *other
 }
}

// 'static lifetime — data lives for the entire program
struct StaticHolder { data: &'static str }

fn main() {
 let x = 42;
 let holder = RefHolder { reference: &x, count: 0 };
 println!("{}", holder.get_ref());
 let sh = StaticHolder { data: "forever" };
}
```

Lifetime Bounds & Constraints

The syntax `'a: 'b` means "`'a` outlives `'b`" — a reference valid for `'a` is also valid for any shorter scope `'b`. This is used in both function signatures and generic type bounds.

```
// 'a: 'b — 'a must outlive 'b
fn longest_with_constraint<'a, 'b>(x: &'a str, y: &'b str) -> &'a str
where 'a: 'b {
    println!("y is: {}", y);
    x
}

// T: 'a — all references inside T must outlive 'a
struct Wrapper<'a, T: 'a> { value: &'a T }

// 'static constraint — T contains no non-static references
fn requires_static<T: 'static>(t: T) { ... }

fn main() {
    requires_static("static str");    // OK — &'static str
    let x = 5;
    // requires_static(&x);           // ERROR — &x is not 'static
}
```

Note: As of Rust 2018, `T: 'a` bounds on struct fields are inferred by the compiler — you rarely need to write them explicitly.

Common Lifetime Mistakes

These are the five most frequent lifetime errors engineers encounter. Understanding them builds intuition for what the borrow checker is actually enforcing.

Mistake 1 — Return reference to local

`fn bad() -> &str { let s = String::from("hi"); &s }` — `s` is dropped at end of function; returned reference would dangle.

Mistake 2 — Ambiguous output lifetime

`fn longest(x: &str, y: &str) -> &str` — compiler cannot determine which input the return borrows from. Add `'a` to both parameters and the return type.

Mistake 3 — Reference outlives the binding

Creating a reference inside a block and using it outside — `r = &x` inside a block, then reading `r` after the block closes.

Mistake 4 — Struct without lifetime parameter

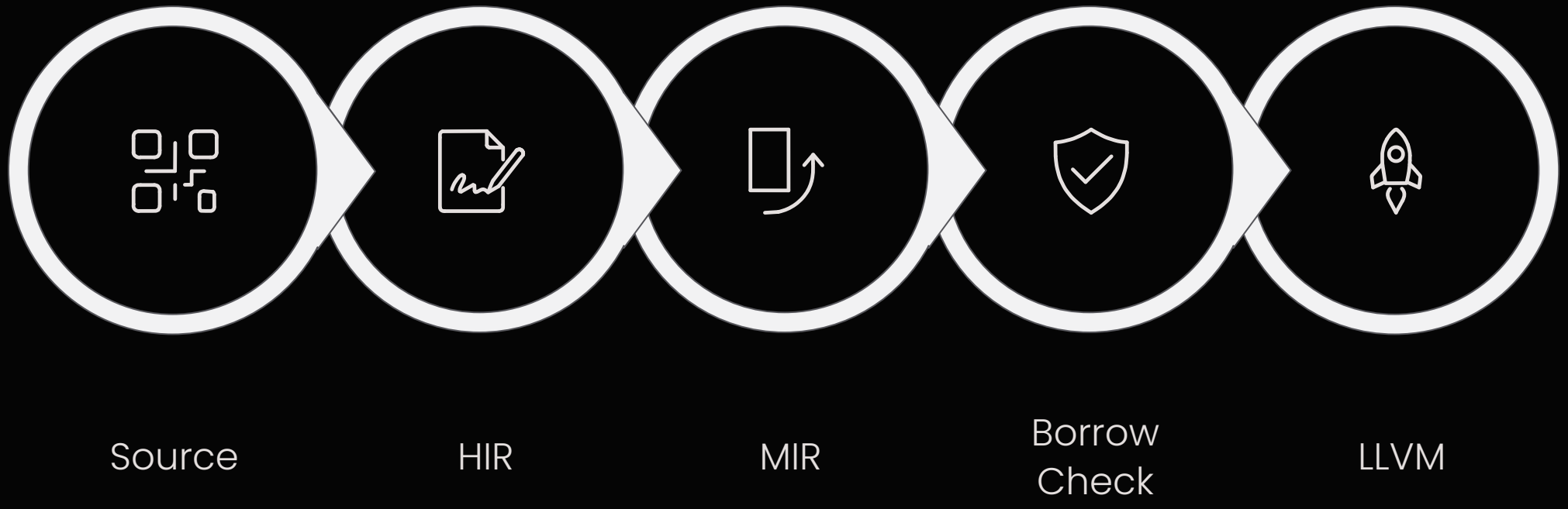
Writing `struct Bad { data: &str }` — the compiler requires `struct Bad<'a> { data: &'a str }`.

Mistake 5 — Lifetime too short in struct usage

Constructing a struct that holds a reference inside a block, then using the struct outside that block after the referent has been dropped.

Compiler Deep Dive — Lifetime Analysis

Lifetime verification happens during the **borrow checking** phase, which operates on the **Mid-level Intermediate Representation (MIR)** — a control-flow graph that makes all lifetimes explicit.



At the MIR stage, all elided lifetimes have been made explicit and each basic block carries precise lifetime region annotations. The borrow checker then models lifetime checking as a **region inference** problem — similar in spirit to type inference — building a constraint graph and verifying it is satisfiable.

How the Borrow Checker Verifies Lifetimes

For each reference, the borrow checker assigns a **lifetime region**, tracks all uses, and solves a constraint system to verify that data outlives every reference to it.

```
fn main() {
    let r;          // ─────┐  lifetime 'r begins
    {               //     │
        let x = 5;  // ─────┼─── lifetime 'x begins
        r = &x;     //     │   borrow: requires 'r ⊆ 'x
    }               // ─────┘    'x ends here!
    println!("{}", r); // ERROR: 'r still active, 'x is gone
}
```

Region Inference

The compiler builds a constraint: `'r ⊆ 'x` (the lifetime of `r` must be contained within the lifetime of `x`). Because `'x` ends before the `println!`, the constraint is violated and the borrow checker reports an error.

Why MIR?

MIR's explicit control-flow graph makes it possible to reason about references across branches, loops, and early returns precisely. Lifetime checking on AST alone would miss many edge cases.

PART 4.3 — ERROR HANDLING

Error Handling Philosophy

Rust eliminates a whole category of bugs by making error handling **explicit and enforced**. There are no unchecked exceptions — errors are values the compiler forces you to acknowledge.

Two Categories of Errors

Rust draws a sharp line between programming bugs (unrecoverable) and expected failures (recoverable). This distinction shapes every API you write.

🔴 Unrecoverable — `panic!`

Used for bugs, violated invariants, and states that should never occur in correct code: out-of-bounds indexing, integer overflow in debug builds, explicit assertion failures. The program terminates immediately (or unwinds the stack).

- Array index out of bounds
- Integer overflow (debug mode)
- `unwrap()` / `expect()` on `None` or `Err`

🟢 Recoverable — `Result<T, E>`

Used for expected, predictable failures that callers should handle: missing files, invalid user input, network timeouts. The function returns a value — `Ok(T)` on success or `Err(E)` on failure — and the caller decides what to do.

- File not found
- Network timeout
- Invalid user input

No exceptions: Rust has no try/catch mechanism at the language level. Every error path is visible in function signatures, making code easier to audit and reason about.

Unrecoverable Errors — `panic!`

When `panic!` fires, Rust prints an error message, begins **unwinding** the stack (calling destructors along the way), and exits. For size- or latency-critical builds, you can switch to `abort` mode.

```
// Explicit panic
fn main() { panic!("crash and burn"); }

// Implicit panics from standard library
let v = vec![1, 2, 3];
v[99];                // panic: index out of bounds

let none: Option<i32> = None;
none.unwrap();        // panic: called unwrap() on None

let err: Result<i32, &str> = Err("oops");
err.expect("Failed to compute value"); // panic with custom message
```

When to Panic

- Examples and prototypes where brevity matters
- Tests — `unwrap()` is idiomatic in test code
- When an error indicates a bug in your code
- When recovery is truly impossible

Abort Mode (Cargo.toml)

```
[profile.release]
panic = "abort"
```

Skips stack unwinding for smaller binaries — useful in embedded or WebAssembly targets where binary size is critical.

How `panic!` Works Under the Hood

Panic Fired

panic! prints message and begins unwind

Stack Unwinds

Drop implementations run for each stack frame

Process Exit

Process exits after cleanup completes

Abort Mode

Skips unwind and terminates immediately

During unwinding, Rust calls `Drop` on every value in each stack frame — ensuring resources (file handles, locks, heap memory) are released even after a panic. `panic::catch_unwind` can intercept panics at a boundary, but this is rare and mainly used in FFI or test harnesses.

```
use std::panic;
let result = panic::catch_unwind(|| {
 panic!("oops!");
});
// result is Err(...) containing panic info
```

Recoverable Errors — `Result<T, E>`

`Result` is a standard enum with two variants: `Ok(T)` carrying a success value and `Err(E)` carrying an error value. The type system forces callers to acknowledge both cases.

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    // Pattern-match on the Result
    let file = match File::open("hello.txt") {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Can't create file: {:?}", e),
            },
            other => panic!("Problem opening file: {:?}", other),
        },
    };
}
```

Closures make this cleaner: The `unwrap_or_else` and `map_err` combinator methods on `Result` allow chaining without deeply nested match expressions — useful when you want functional-style error handling.

Rust vs C vs C++ Error Handling

One of Rust's most significant improvements over systems languages is making error handling *impossible to accidentally ignore*. Compare the three approaches:

C — Return Codes

Errors signalled via `NULL` pointers or `errno`. Forgetting to check leads to **undefined behaviour**. Nothing in the language enforces that you handle errors.

C++ — Exceptions

Exceptions can propagate from any call site invisibly. The compiler does not require callers to declare or handle them. Exception paths are notoriously hard to audit in large codebases.

Rust — `Result<T, E>`

Errors are values in the type system. The compiler emits a warning if you ignore a `Result`, and patterns like `?` make propagation ergonomic without hiding the error path.

Propagating Errors with `?`

The `?` operator is the idiomatic way to propagate errors upward. It either unwraps the `Ok` value and continues, or returns the `Err` immediately — converting the error type via `From` if needed.

Verbose — Without ?

```
fn read_username() -> Result<String, io::Error> {
    let file_result = File::open("hello.txt");
    let mut file = match file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };
    let mut username = String::new();
    match file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

Idiomatic — With ?

```
fn read_username() -> Result<String, io::Error> {
 let mut file = File::open("hello.txt")?;
 let mut username = String::new();
 file.read_to_string(&mut username)?;
 Ok(username)
}

// Even shorter — method chaining
fn read_username_short() -> Result<String, io::Error> {
 let mut username = String::new();
 File::open("hello.txt")?
 .read_to_string(&mut username)?;
 Ok(username)
}

// Shortest — standard library helper
fn read_username_std() -> Result<String, io::Error> {
 fs::read_to_string("hello.txt")
}
```

How ? Actually Works

The ? operator is syntactic sugar that expands to a match expression with an automatic type conversion via the From trait — enabling seamless composition across different error types.

```
// This:
let x = some_result?;

// Expands to:
let x = match some_result {
 Ok(val) => val,
 Err(err) => return Err(From::from(err)), // error type conversion!
};

// ? also works with Option
fn bar() -> Option<i32> {
 let x = some_option?; // returns None if some_option is None
 Some(x * 2)
}

// Mixing error types with Box<dyn Error>
fn read_and_parse() -> Result<i32, Box<dyn std::error::Error>> {
 let mut s = String::new();
 io::stdin().read_line(&mut s)?; // io::Error → Box<dyn Error>
 let n: i32 = s.trim().parse()?; // ParseIntError → Box<dyn Error>
 Ok(n)
}
```

Defining Custom Error Types

For any non-trivial library, defining a custom error enum gives callers the ability to pattern-match on specific failure modes and provides rich diagnostic information.

```
use std::{fmt, error::Error};

#[derive(Debug)]
pub enum AppError {
 Io(std::io::Error),
 Parse(std::num::ParseIntError),
 Validation(String),
}

impl fmt::Display for AppError {
 fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
 match self {
 AppError::Io(e) => write!(f, "IO error: {}", e),
 AppError::Parse(e) => write!(f, "Parse error: {}", e),
 AppError::Validation(s) => write!(f, "Validation error: {}", s),
 }
 }
}

impl Error for AppError {
 fn source(&self) -> Option<&(dyn Error + 'static)> {
 match self {
 AppError::Io(e) => Some(e),
 AppError::Parse(e) => Some(e),
 _ => None,
 }
 }
}
```

Converting Between Error Types with `From`

Implementing `From<OtherError> for AppError` enables the `?` operator to automatically convert foreign error types into your own — keeping call sites clean.

```
impl From<io::Error> for AppError {
    fn from(error: io::Error) -> Self { AppError::Io(error) }
}
impl From<ParseIntError> for AppError {
    fn from(error: ParseIntError) -> Self { AppError::Parse(error) }
}
impl From<String> for AppError {
    fn from(message: String) -> Self { AppError::Message(message) }
}

// Now ? converts automatically!
fn read_number_from_file() -> Result<i32, AppError> {
    let mut file = File::open("number.txt")?;     // io::Error → AppError
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;          // io::Error → AppError
    let num = contents.trim().parse()?;           // ParseIntError → AppError
    Ok(num)
}
```

Tip: The `thiserror` crate generates these `From` impls and `Display` implementations via a derive macro, eliminating the boilerplate entirely.

The `anyhow` Crate — For Applications

`anyhow` provides a single opaque error type that can wrap *any* error, making it ideal for application-level code where you want ergonomic propagation with rich context rather than pattern-matching on variants.

```
use anyhow::{Result, anyhow, Context};
use std::fs;

fn read_config() -> Result<String> {
 let content = fs::read_to_string("config.json")
 .with_context(|| "Failed to read config file")?;
 if content.is_empty() {
 return Err(anyhow!("Config file is empty"));
 }
 Ok(content)
}

fn process_data() -> Result<()> {
 let data = fs::read_to_string("data.txt")
 .context("reading data file")?;
 let parsed: i32 = data.trim().parse()
 .context("parsing data as integer")?;
 println!("Data: {}", parsed);
 Ok(())
}

fn main() -> Result<()> {
 match process_data() {
 Ok(_) => println!("Success!"),
 Err(e) => eprintln!("Error: {:?}", e), // prints full context chain
 }
 Ok(())
}
```

The `thiserror` Crate — For Libraries

`thiserror` uses proc-macro derives to eliminate the boilerplate of `Display` implementations and `From` conversions — keeping your error types expressive without being verbose.

```
use thiserror::Error;

#[derive(Error, Debug)]
pub enum DataStoreError {
 #[error("data store disconnected")]
 Disconnect(#[from] io::Error), // generates From<io::Error>

 #[error("redaction failed on `{0}`")]
 Redaction(String),

 #[error("invalid header (expected {expected:?}, found {found:?})")]
 InvalidHeader { expected: String, found: String },

 #[error("unknown data store error")]
 Unknown,
}

fn read_user_data() -> Result<UserData, DataStoreError> {
 let file = File::open("users.db")?; // io::Error auto-converted
 if !validate_header(&file) {
 return Err(DataStoreError::InvalidHeader {
 expected: "USERv1".into(),
 found: "UNKNOWN".into(),
 });
 }
 // ...
}
```

Error Handling Decision Tree

When deciding how to signal an error from a function, follow this decision process. The right choice depends on whether the caller can meaningfully recover, and on the context — library vs application code.

- Use `Result` for expected, predictable failures that callers should handle.
- Use `panic!` for bugs, violated preconditions, or states that indicate programmer error.
- Use `thiserror` in libraries — gives downstream users structured error variants to match on.
- Use `anyhow` in applications — prioritises ergonomic propagation and rich context chains over matchable variants.

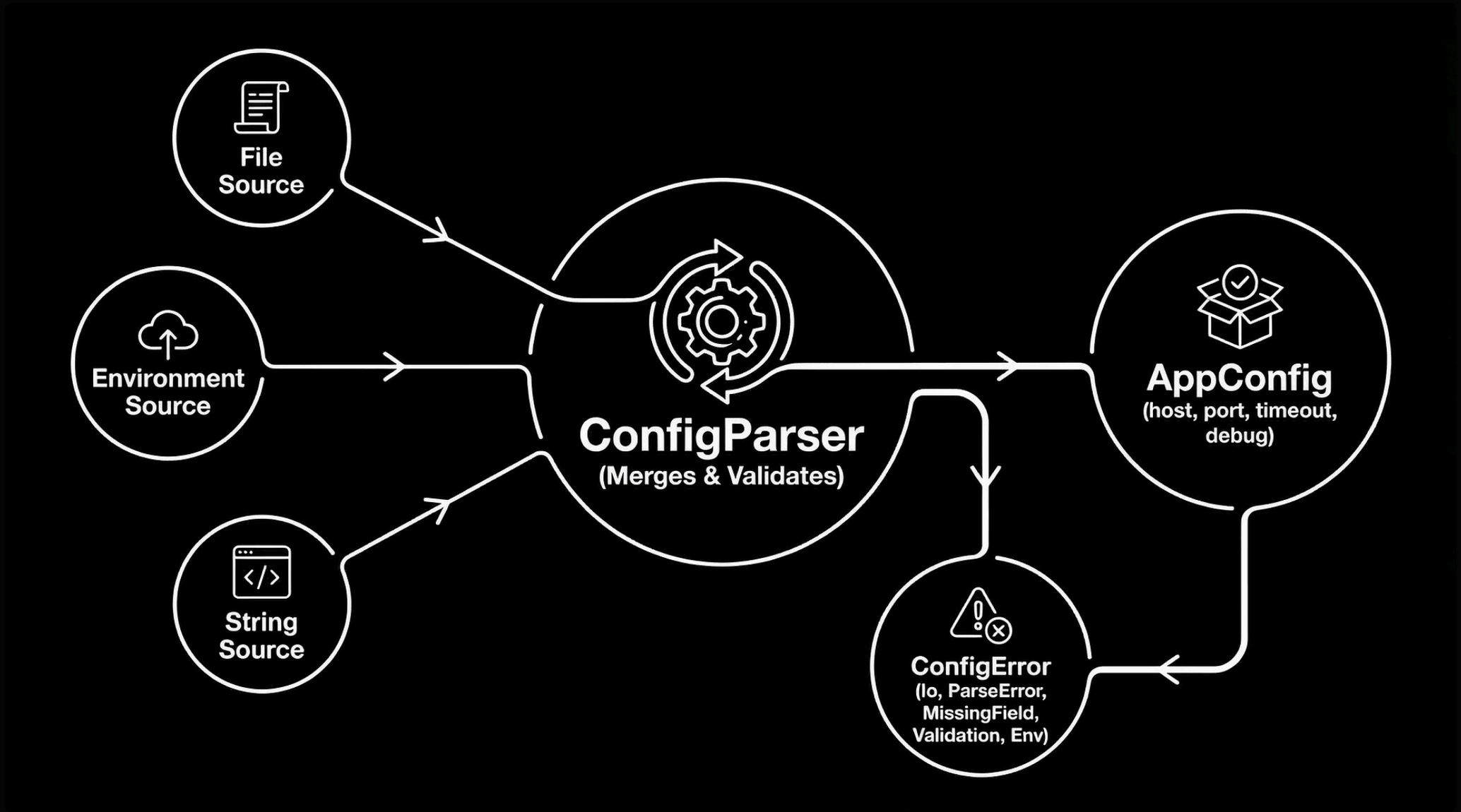
PART 4.4 — LAB SESSION

Lab Session — Build a Configuration Parser

In this lab you will apply traits, generics, lifetimes, and error handling to build a real, reusable **configuration parser library** in Rust.

Lab Overview — Goals & Architecture

The library reads configuration values from multiple pluggable **sources** (files, environment variables, raw strings), merges them in priority order, validates the values, and converts them into a strongly-typed `AppConfig` struct.



Lab Setup — Data Structures

Start by defining the configuration struct and a rich error enum. The error enum uses `thiserror` to derive `Display` and `From` implementations automatically.

```
use std::collections::HashMap;
use thiserror::Error;

#[derive(Debug, PartialEq)]
struct AppConfig {
 host: String,
 port: u16,
 timeout: u64,
 debug: bool,
}

#[derive(Error, Debug)]
enum ConfigError {
 #[error("IO error: {0}")]
 Io(#[from] std::io::Error),

 #[error("Parse error for field '{field}': {message}")]
 ParseError { field: String, message: String, value: String },

 #[error("Missing required field: {0}")]
 MissingField(String),

 #[error("Validation error: {0}")]
 Validation(String),

 #[error("Environment error: {0}")]
 Env(#[from] std::env::VarError),
}
```

Defining the `ConfigSource` Trait

The `ConfigSource` trait abstracts over any data source that can produce a flat key-value map. This design makes the parser open for extension — new sources require only a new `impl` block.

```
trait ConfigSource {
 fn read(&self) -> Result<HashMap<String, String>, ConfigError>;
 fn name(&self) -> &'static str;
}

struct FileSource { path: String }

impl ConfigSource for FileSource {
 fn read(&self) -> Result<HashMap<String, String>, ConfigError> {
 let content = std::fs::read_to_string(&self.path)?;
 let mut map = HashMap::new();
 for line in content.lines() {
 if line.trim().is_empty() || line.starts_with('#') { continue; }
 if let Some((key, value)) = line.split_once('=') {
 map.insert(key.trim().to_string(), value.trim().to_string());
 }
 }
 Ok(map)
 }
 fn name(&self) -> &'static str { "file" }
}
```

Environment & String Sources

The environment source strips a configurable prefix from environment variable names and lower-cases the keys. The string source is invaluable for unit testing — no filesystem required.

```
struct EnvSource { prefix: String }

impl ConfigSource for EnvSource {
 fn read(&self) -> Result<HashMap<String, String>, ConfigError> {
 let mut map = HashMap::new();
 for (key, value) in std::env::vars() {
 if key.starts_with(&self.prefix) {
 let config_key = key[self.prefix.len()..].to_lowercase();
 map.insert(config_key, value);
 }
 }
 Ok(map)
 }
 fn name(&self) -> &'static str { "environment" }
}

struct StringSource { content: String }

impl ConfigSource for StringSource {
 fn read(&self) -> Result<HashMap<String, String>, ConfigError> {
 let mut map = HashMap::new();
 for line in self.content.lines() {
 if line.trim().is_empty() || line.starts_with('#') { continue; }
 if let Some((key, value)) = line.split_once('=') {
 map.insert(key.trim().to_string(), value.trim().to_string());
 }
 }
 Ok(map)
 }
 fn name(&self) -> &'static str { "string" }
}
```

The Parser — Merging Sources

The `ConfigParser` collects trait objects in a `Vec<Box<dyn ConfigSource>>`, reads each in order (later sources override earlier ones), and attempts type-safe conversion into `AppConfig`.

```
struct ConfigParser { sources: Vec<Box<dyn ConfigSource>> }

impl ConfigParser {
 fn new() -> Self { Self { sources: Vec::new() } }
 fn add_source(&mut self, source: Box<dyn ConfigSource>) {
 self.sources.push(source);
 }
 fn parse(&self) -> Result<AppConfig, ConfigError> {
 let mut config_map = HashMap::new();
 for source in &self.sources {
 match source.read() {
 Ok(values) => {
 for (k, v) in values { config_map.insert(k, v); }
 }
 Err(e) => eprintln!("Warning from {}: {}", source.name(), e),
 }
 }
 Ok(AppConfig {
 host: self.parse_string(&config_map, "host")?,
 port: self.parse_u16(&config_map, "port")?,
 timeout: self.parse_u64(&config_map, "timeout")?,
 debug: self.parse_bool(&config_map, "debug").unwrap_or(false),
 })
 }
}
```

Helper Parsing Methods

Each helper retrieves a key from the merged map, returning a structured `ConfigError` variant that carries the field name and the offending value — making error messages actionable for end users.

```
impl ConfigParser {
 fn parse_string(&self, map: &HashMap<String,String>, key: &str)
 -> Result<String, ConfigError>
 {
 map.get(key).cloned()
 .ok_or_else(|| ConfigError::MissingField(key.to_string()))
 }

 fn parse_u16(&self, map: &HashMap<String,String>, key: &str)
 -> Result<u16, ConfigError>
 {
 let value = map.get(key)
 .ok_or_else(|| ConfigError::MissingField(key.to_string()))?;
 value.parse::<u16>()
 .map_err(|_| ConfigError::ParseError {
 field: key.to_string(),
 message: format!("Invalid u16: '{}'", value),
 value: value.clone(),
 })
 }

 fn parse_bool(&self, map: &HashMap<String,String>, key: &str)
 -> Option<bool>
 {
 map.get(key).and_then(|v| match v.to_lowercase().as_str() {
 "true" | "yes" | "1" => Some(true),
 "false" | "no" | "0" => Some(false),
 _ => None,
 })
 }
}
```

Main Function — Wiring It Together

The `main` function demonstrates the complete usage: add sources in priority order (file first, then environment to allow overrides), parse, and print a detailed error chain if anything goes wrong.

```
fn main() -> Result<(), ConfigError> {
 let mut parser = ConfigParser::new();
 parser.add_source(Box::new(FileSource {
 path: "config.txt".to_string(),
 }));
 parser.add_source(Box::new(EnvSource {
 prefix: "APP_".to_string(),
 }));

 match parser.parse() {
 Ok(config) => {
 println!("Config loaded:");
 println!(" host: {}", config.host);
 println!(" port: {}", config.port);
 println!(" timeout: {}", config.timeout);
 println!(" debug: {}", config.debug);
 Ok(())
 }
 Err(e) => {
 eprintln!("Failed to load config: {}", e);
 let mut source = e.source();
 while let Some(err) = source {
 eprintln!(" caused by: {}", err);
 source = err.source();
 }
 Err(e)
 }
 }
}
```

Lab Challenge Tasks

The lab is structured in three difficulty tiers — complete whichever levels suit your current comfort with Rust. Each tier builds directly on the previous.

🟢 Level 1 — Basic Implementation

Implement all `parse_*` helper methods. Create a sample `config.txt` file. Verify that missing required fields produce clear error messages. Write a basic integration test using `StringSource`.

🟡 Level 2 — Enhancements

Add a JSON source using `serde_json`. Implement config validation (port in 1–65535, non-empty hostname). Add default values so optional fields do not trigger `MissingField`.

🔴 Level 3 — Advanced Features

Implement hot-reloading using `notify` (file-watcher crate). Add support for nested config sections. Create a derive macro that generates parse logic for arbitrary config structs. Add encryption for sensitive values.

Traits — Key Takeaways

Shared Behaviour

Traits define required and default methods across unrelated types — Rust's answer to interfaces, but more powerful.

Orphan Rule

At least one of trait or type must be local to your crate — guarantees global coherence.

Dispatch Choice

Static dispatch via generics: zero overhead, larger binary. Dynamic dispatch via `dyn Trait`: flexible, small vtable cost.

Operator Overloading

All operator behaviour is defined through `std::ops` traits — no special syntax needed.

Lifetimes — Key Takeaways

1

References Are Tracked

Every reference carries a lifetime — the compiler ensures data outlives every reference to it. No garbage collector, no runtime cost.

2

Elision Covers 90% of Cases

Three simple elision rules mean you rarely need to write lifetime annotations on ordinary functions and methods.

3

Structs That Hold References

A struct holding a reference must declare a lifetime parameter and cannot outlive the referent — this is expressed entirely in the type signature.

4

MIR Borrow Checking

Lifetime verification is performed on the MIR control-flow graph using region inference — the same fundamental idea as type inference, applied to memory scopes.

Error Handling — Key Takeaways

`panic!` for Bugs

Reserve `panic!` for programmer errors and violated invariants. It is not a general-purpose error propagation tool.

`?` Propagates Cleanly

The `?` operator makes error propagation ergonomic without hiding it — errors remain visible in function signatures.

`Result<T,E>` for Expected Failures

Model all recoverable errors as values. The compiler ensures callers acknowledge them — no silent failures.

Libraries vs Applications

Use `thiserror` for structured library errors; use `anyhow` for application-level error context chains.

The Bigger Picture — Why These Features Matter

Traits, lifetimes, and error handling are not isolated features — they work together to deliver Rust's core promise: systems-level performance with memory safety guaranteed at compile time.

Zero-Cost Abstractions

Trait-based generics with static dispatch mean you pay *nothing* at runtime for polymorphism. The same code that works on any `Iterator` compiles to machine code as efficient as hand-written loops.

No Garbage Collector

Lifetimes allow the compiler to insert exactly the right cleanup code at the right point — deterministic resource management without a runtime GC pause.

Fearless Concurrency

The same borrow checker that enforces lifetimes prevents data races at compile time — the `Send` and `Sync` marker traits (both object-safe trait examples) encode thread-safety directly in the type system.

Composable Error Handling

Because errors are values, they compose naturally with generics, iterators, and async — you get the full power of Rust's type system applied to failure paths.

Trait System — Quick Reference Card

Feature	Syntax	Use When
Define trait	`trait Foo { fn bar(&self); }`	Shared behaviour across types
Implement trait	`impl Foo for MyType { ... }`	Making a type conform to an interface
Trait bound	`fn f<T: Foo>(t: &T)`	Generic functions constrained to trait
impl Trait (arg)	`fn f(t: &impl Foo)`	Simple one-off trait bound in args
impl Trait (ret)	`fn f() -> impl Foo`	Return opaque type without naming it
Trait object	`&dyn Foo` / `Box<dyn Foo>`	Heterogeneous collections, dynamic dispatch
Supertrait	`trait Bar: Foo { ... }`	Trait requires another trait
Associated type	`type Item;`	One output type per implementation
Operator overload	`impl ops::Add for T { ... }`	Custom arithmetic / indexing behaviour

Lifetime & Error Quick Reference

Lifetime Cheat Sheet

Syntax	Meaning
`'a`	Named lifetime parameter
`&'a T`	Reference valid for at least `'a`
`'a: 'b`	`'a` outlives `'b`
`'static`	Lives for the entire program
`T: 'a`	All refs in T outlive `'a`

Error Handling Cheat Sheet

Tool	Use For
`panic!`	Bugs, invariant violations
`Result<T,E>`	Recoverable failures
`?`	Propagate + convert errors
`thiserror`	Library custom error types
`anyhow`	Application error context

Preview — Module 5: Modules, Packaging & Advanced Cargo

In Module 5, we move from language features to **project structure and the Rust ecosystem** — learning how to organise large codebases and share your work with the world.

Modules & Visibility

Organising code with `mod`, controlling visibility with `pub`, and the filesystem module layout conventions.

Cargo In Depth

Workspaces, build profiles, custom build scripts (`build.rs`), dependency features and optional deps.

Publishing Crates

Documentation with `rustdoc`, semantic versioning, and publishing to `crates.io`.

Homework: Complete the lab session and experiment with at least two different error handling patterns (`anyhow` and a custom error enum) in your own code before the next session.

Additional Resources

📖 Official Documentation

- **The Rust Book** — Chapters 10 & 19 (Traits & Lifetimes), Chapter 9 (Error Handling)
- **Rust Reference** — Traits, Lifetime elision, Object safety
- **std::error** — The standard error trait and its contract

📦 Crates to Know

- `thiserror` — Derive macros for library error types
- `anyhow` — Flexible, context-rich errors for applications
- `snafu` — Alternative error handling with location context
- `miette` — Fancy diagnostic error reporting

🔬 Deep Dives

- **"Error Handling in Rust"** — Andrew Gallant (BurntSushi) blog
- **"Lifetimes and Regions"** — Niko Matsakis's blog series
- **"Rust Design Patterns"** — Trait object patterns and idioms
- **Rustonomicon** — Unsafe Rust and lifetime variance deep dive

Questions & Discussion

"Rust gives you the tools to express your intentions clearly in the type system — and then holds you to them."

Traits still fuzzy?

Ask about static vs dynamic dispatch, object safety, or supertrait design.

Lifetimes confusing?

Bring your compiler error — we'll walk through the borrow checker's reasoning together.

Error handling patterns?

When to use `panic!` vs `Result`, and how to structure error types for your project.

Lab questions?

Stuck on a challenge level or curious about extending the config parser further?

Chandrashekar Babu · **training@chandrashekar.info**