Rust Fundamentals

A Complete 8-Chapter Programming Course

Topics covered:

cargo & the toolchain · Variables & basic types

Ownership & borrowing · Structs & methods

Enums & pattern matching · Error handling · Collections & iterators

Exercises: 24 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · framed throughout against Go

Course 1 of 3 · Intermediate and Advanced courses follow

Table of Contents

01 Getting Started: cargo, rustc, and Your First Program

02 Variables & Basic Types

03 Ownership

04 Borrowing & References

05 Structs & Methods

06 Enums & Pattern Matching

07 Error Handling

08 Collections & Iterators

CHAPTER 1 OF 8

Getting Started: cargo, rustc, and Your First Program

COURSE 1 · CH 1

Getting Started: cargo, rustc, and Your First Program

A compiled systems language built around one goal: memory safety without a garbage collector

Rust is a **compiled, systems-level** language, in the same broad family as Go — source code is built into a standalone executable, not interpreted line by line. But the two languages exist for genuinely different reasons, and that difference shapes everything in this course.

Why Rust Exists

Go, JavaScript, Python, Ruby, PHP, and Kotlin all rely on a **garbage collector** — a background process that automatically frees memory the program no longer needs. This is convenient, but it comes with real costs: runtime overhead, and pause times that are hard to predict exactly when they'll happen. Older systems languages like C and C++ skip the garbage collector entirely, giving the programmer full manual control — but with it, real risk: dangling pointers, buffer overflows, use-after-free bugs that have caused decades of security vulnerabilities.

Rust's entire reason for existing is refusing to accept that trade-off: manual-control-level performance and predictability, **without** the memory-safety bugs manual control usually invites — enforced not at runtime, but by the compiler itself, before the program ever runs. Chapters 3–4 cover the actual mechanism (ownership and borrowing); this chapter is just naming the goal everything else in this course serves.

Installing Rust: rustup and rustc

rustup is Rust's official installer and toolchain manager — it installs and manages Rust versions, similar in spirit to a version manager, but built and maintained by the Rust team itself as the standard, recommended way to get Rust at all.

```
# after installing via rustup:
rustc --version
# rustc 1.76.0 (07dca489a 2024-02-04)
```

rustc is the raw compiler underneath everything — the direct equivalent of what runs behind the scenes when Go's `go build` compiles a file. In practice, most day-to-day work goes through **cargo** instead of calling

`rustc` directly.

cargo: Rust's Build Tool & Package Manager

This is the first real toolchain difference from Go. Go's toolchain is deliberately minimal — `go run` and `go build` are close to the whole story. **cargo** bundles far more into one official tool from day one: building, running, dependency management against **crates.io** (Rust's package registry), and testing — closer in spirit to npm and a bundler combined, except built directly into Rust's own official tooling rather than assembled from separate third-party choices.

```
cargo new hello_rust
# creates:
#   hello_rust/
#     Cargo.toml   (the project manifest — name, version, dependencies)
#     src/main.rs  (your actual code)

cd hello_rust
cargo run
```

`Cargo.toml` is the direct equivalent of Go's `go.mod` — but where a `go.mod` stays minimal, `Cargo.toml` is the central place dependencies, metadata, and build configuration all live together.

The Smallest Rust Program

```
fn main() {
    println!("Hello, Rust!");
}
```

`fn main()` is the entry point — the same role `func main()` plays in Go; execution always starts here. `println!` is Rust's equivalent of `fmt.Println`, but notice the `!` — that marks it as a **macro**, not an ordinary function call. Macros are a genuinely different Rust concept from anything in Go or JavaScript (Course 3's own Macros chapter covers writing them); for now, just recognize that `!` after a name means "this is a macro," and `println!` is the one used constantly throughout this course.

CONCEPT	GO	RUST
Compile & run in one step	`go run file.go`	`cargo run`
Produce a standalone binary	`go build`	`cargo build --release`
Project manifest	`go.mod (minimal)`	`Cargo.toml (build config + dependencies)`
Print a line	`fmt.Println(...)`	`println!(...)`

CONCEPT	GO	RUST
Entry point	`func main()`	`fn main()`

WHY MEMORY SAFETY KEEPS COMING UP

Every chapter in this course's first half builds toward the same destination: how Rust guarantees memory safety at compile time, with zero runtime garbage collector. Chapter 3's ownership model is the actual mechanism — this chapter is just establishing why that mechanism is worth learning in the first place.

SEMICOLONS ARE MEANINGFUL, NOT OPTIONAL STYLE

Unlike idiomatic Go (which omits semicolons via automatic insertion) and unlike JavaScript (where they're often optional), Rust genuinely distinguishes a **statement** (ending in `;`) from an **expression** (no semicolon, and its value can be returned). Inside `main` right now this mostly just means "don't forget the semicolon" — but this exact distinction becomes meaningful later when a function's last line, with no semicolon, becomes its return value. Missing or extra semicolons produce real compiler errors, not silent behavior differences.

Coding Challenges

CHALLENGE 1

Use cargo new to create a project called greeting, then edit src/main.rs so it prints your name and a short greeting using two separate println! calls. Run it with cargo run.

View solution

CHALLENGE 2

Write a program that prints "Year: 2026" and "Language: Rust" using println! with placeholders ({}) instead of concatenating strings — two separate println! calls, each using a placeholder.

View solution

CHALLENGE 3

Explain, in your own words, why Rust's pitch ("manual-control performance without manual-control memory bugs") is a genuinely different trade-off than either a garbage-collected language or C/C++ — and name the two chapters coming up that actually deliver on that promise.

View solution

CHAPTER 1 QUICK REFERENCE

- **Rust's core goal:** memory safety without a garbage collector, enforced at compile time
- **rustup** — the official installer/toolchain manager; **rustc** — the raw compiler underneath
- **cargo** — Rust's official build tool + package manager + test runner, bundled together (unlike Go's minimal toolchain)
- **Cargo.toml** — the project manifest (dependencies, build config) — Rust's fuller-featured counterpart to Go's `go.mod`
- **cargo new / cargo run / cargo build --release** — create, run, and produce a standalone binary
- **fn main() { }** — the entry point, same role as Go's `func main()`
- **println!(...)** — note the `!`: this is a macro, not an ordinary function
- Semicolons distinguish statements from expressions — a distinction that becomes load-bearing later, not just style
- **Next chapter:** variables, basic types, immutability by default (`let` vs `let mut`), and shadowing

CHAPTER 2 OF 8

Variables & Basic Types

COURSE 1 · CH 2

Variables & Basic Types

Immutable by default, explicit scalar types, and a genuinely different kind of "redeclaring a variable"

Chapter 1 named memory safety as Rust's whole reason for existing. This chapter's first rule is a direct expression of that philosophy: variables are **immutable by default** — the opposite of what Go, JavaScript, and Python all do.

Immutability by Default

```
let x = 5;
x = 6; // compile error: cannot assign twice to immutable variable
```

`let x = 5;` creates an **immutable** binding — attempting to reassign it is a **compile error**, not something that fails at runtime or silently succeeds. Mutability has to be opted into explicitly:

```
let mut x = 5;
x = 6; // fine — x was declared mutable
```

This is a deliberate design choice, not an arbitrary restriction: most variables, in practice, are never reassigned after being set — making immutability the default means the compiler can catch an accidental reassignment as a mistake, rather than a program silently doing something the programmer didn't intend.

Scalar Types

Rust's basic scalar types are explicit about size and signedness, unlike Go's simpler `int` / `uint` or JavaScript's single `Number` type:

```rust
let a: i32 = -10;      // signed 32-bit integer
let b: u32 = 10;       // unsigned 32-bit integer (no negatives)
let c: f64 = 3.14;     // 64-bit floating point (the default for decimals)
let d: bool = true;
let e: char = 'R';      // a single Unicode scalar value, 4 bytes — closer to Go's rune than a 1-byte ch
```

Compound Types

A **tuple** groups a fixed number of values of *mixed* types; an **array** holds a fixed number of values of the *same* type. Both have a size fixed at compile time — a dynamically growable list is `Vec<T>`, covered in Chapter 8.

```rust
let point: (i32, f64, char) = (3, 9.5, 'Z');
let x = point.0;  // tuples are indexed with .0, .1, .2

let scores: [i32; 5] = [90, 85, 78, 92, 88];
let first = scores[0];
```

Type Inference & Explicit Annotations

Rust infers types in most cases — `let x = 5;` is inferred as `i32` without an annotation, the same convenience Go's `:=` provides. An explicit annotation is only needed when the inferred default isn't what's wanted, or when the type genuinely can't be inferred from context alone.

```rust
let x = 5;             // inferred as i32
let y: u8 = 5;        // explicitly u8 instead
```

Shadowing

Rust allows re-declaring a variable with the **same name** using `let` again — this creates a brand-new binding that **shadows** the previous one, and can even change its type entirely. This is genuinely different from Go, where redeclaring a name in the same scope with `:=` is usually a compile error unless at least one new variable is being introduced.

```rust
let input = "42";
let input: i32 = input.parse().unwrap(); // shadowed — now a number, same name
```

The second `let input` doesn't mutate the original string variable — it creates a separate variable that happens to share the name, and the earlier `&str` version is no longer accessible from this point in the scope

onward.

CONCEPT	GO	RUST
Default mutability	`Mutable by default`	`Immutable by default (mut opts in)`
Type inference	`:= infers the type`	`let infers the type`
Redeclaring the same name	`Compile error (with :=, unless a new var is added)`	`Allowed — shadowing, can change type`
Fixed-size list	`Array (rarely used directly)`	`[T; N] — fixed-size, same type`

IMMUTABILITY IS THE SAME SAFETY PHILOSOPHY FROM CHAPTER 1

Defaulting to immutable bindings is a small, early instance of the same idea Chapter 3's ownership model takes much further: the compiler catches an entire category of mistakes (accidental mutation) before the program ever runs, rather than leaving it to be discovered at runtime or in production.

SHADOWING IS NOT MUTATION — AND IT DOESN'T SURVIVE THE SCOPE

Shadowing requires using `let` again each time — a plain reassignment (`input = ...` with no `let`) is mutation, and requires `mut`, an entirely different mechanism. Shadowing is also scope-limited: once a shadowed variable's block ends, the *outer* variable (if one exists) becomes visible again, unaffected by whatever the inner shadow did. Confusing shadowing with mutation is a common early mix-up — they look superficially similar but follow completely different rules.

Coding Challenges

CHALLENGE 1

Declare an immutable variable holding your age as a u8, then write a second line that attempts to reassign it. Run cargo run, note the exact compiler error, then fix it using mut.

View solution

CHALLENGE 2

Create a tuple holding a product's id (i32), price (f64), and category initial (char). Print each field individually using tuple indexing (.0, .1, .2).

View solution

CHALLENGE 3

Using shadowing, take a variable holding the string "100", and shadow it twice: first into an i32, then into that number multiplied by 2. Print the final value, and explain why this required let each time rather than plain reassignment.

View solution

CHAPTER 2 QUICK REFERENCE

- **let x = ...** — immutable by default; reassigning without `mut` is a compile error
- **let mut x = ...** — explicitly opts into mutability
- **Scalar types:** i32/u32/i64/u64 (explicit width + signedness), f32/f64, bool, char (4-byte Unicode scalar)
- **Tuples** — fixed size, mixed types, indexed with `.0` / `.1` / `.2`
- **Arrays** — fixed size, same type, `[T; N]`; a growable list is `Vec<T>` (Ch.8)
- **Shadowing** — re-declaring with `let` creates a new binding, can change type, scope-limited
- Shadowing ≠ mutation — shadowing needs `let` each time; mutation needs `mut`, no new `let`
- **Next chapter:** Ownership — move semantics, one owner at a time, and why this eliminates the need for a garbage collector

CHAPTER 3 OF 8

Ownership

COURSE 1 · CH 3

Ownership

The mechanism that actually delivers on Chapter 1's promise — memory safety with no garbage collector

Chapter 1 promised memory safety without a garbage collector. This chapter is where that promise becomes concrete — **ownership** is Rust's single defining idea, and nearly everything else in this course builds on it.

The Three Ownership Rules

1. Each value has exactly **one owner** at a time.
2. When the owner goes out of scope, the value is **dropped** (its memory freed) automatically.
3. Ownership can be **transferred** (moved), but a value is never implicitly duplicated.

Every example in this chapter is really just these three rules playing out in different situations.

Stack vs. Heap, Briefly

Simple fixed-size types (`i32`, `bool`, `char`, tuples of these) live entirely on the **stack** — cheap to copy, since copying just means duplicating a few fixed bytes. A `String` (or `Vec`, covered in Chapter 8) stores its actual character data on the **heap**, with only a pointer, length, and capacity sitting on the stack. This distinction is exactly why move semantics matter: moving a `String` is cheap — it's just copying that small pointer/length/capacity trio, never the underlying heap data itself.

Move Semantics

```
let s1 = String::from("hello");
let s2 = s1;
println!("{}", s1); // compile error: value borrowed after move
```

Coming from Go or JavaScript, `let s2 = s1` looks like it should leave *both* variables usable — either both reference the same underlying string, or the whole value gets copied. Rust does neither: it **moves** ownership

of the heap data from `s1` to `s2`, and `s1` becomes **invalid** from that point on. Using it afterward is a compile error, not a runtime surprise.

Why Not Just Copy Everything?

The alternative — deep-copying on every assignment — would be safe, but expensive, and that cost would be invisible in the code. Rust instead makes **moving** the default (cheap, just a pointer/length/capacity copy) and requires an **explicit** `.clone()` call whenever a real, expensive deep copy is actually wanted — making that cost visible right in the source.

```
let s1 = String::from("hello");
let s2 = s1.clone(); // an explicit, real deep copy — both s1 and s2 are valid
println!("{} {}", s1, s2); // fine
```

Simple stack-only types (`i32`, `f64`, `bool`, `char`, and tuples made entirely of these) implement Rust's `Copy` trait instead — copying them is so cheap that Rust just does it automatically on assignment, and *neither* variable becomes invalid:

```
let x = 5;
let y = x;
println!("{} {}", x, y); // fine — i32 is Copy, x was never moved
```

One Owner at a Time & Automatic Cleanup

When a variable goes out of scope — the end of a block or function — Rust automatically calls `drop` on it, freeing its memory immediately and **deterministically**. This is the actual mechanism that eliminates the need for a garbage collector: because ownership rules guarantee, at *compile time*, that exactly one thing is ever responsible for a given piece of data, the compiler always knows the exact moment cleanup should happen — no background process needs to figure it out at runtime.

Ownership and Functions

Passing a value to a function **moves** it in (unless the type is `Copy`) — the function becomes the new owner, and the caller's variable becomes invalid after the call, unless the function explicitly hands ownership back by returning the value.

```rust
fn takes_ownership(s: String) {
    println!("{}", s);
} // s is dropped here — the function's scope ends

fn main() {
    let s = String::from("hello");
    takes_ownership(s);
    println!("{}", s); // compile error: s was moved into the function
}
```

This is genuinely inconvenient if the caller just wanted the function to *look* at the value without giving it up entirely — which is exactly the problem Chapter 4's **borrowing** exists to solve.

	MOVE TYPES	COPY TYPES
Examples	`String, Vec<T>, most custom structs`	`i32, f64, bool, char, tuples of these`
On assignment	`Ownership moves; original becomes invalid`	`Value is copied; both remain valid`
To get a real duplicate	`Explicit .clone()`	`Automatic — no clone needed`

THIS IS THE ANSWER TO CHAPTER 1'S QUESTION

"How does Rust get memory safety without a garbage collector?" — this chapter is the full answer: exactly one owner, enforced at compile time, with deterministic cleanup the instant that owner goes out of scope. No runtime process ever needs to guess when memory is safe to free.

"BORROW OF MOVED VALUE" IS THE MOST COMMON EARLY RUST ERROR

Trying to use a variable after it's been moved — into another variable or into a function — is by far the most frequent compiler error newcomers hit, often described as "fighting the borrow checker." It's not a bug in your understanding; it's the compiler correctly enforcing rule #1. Chapter 4's borrowing (`&` and `&mut`) is specifically designed to solve the exact frustration this chapter's function example just demonstrated — letting a function use a value without taking ownership of it at all.

Coding Challenges

CHALLENGE 1

Write code that creates a String, moves it into a second variable, then attempts to print the original variable. Run cargo run, note the exact compiler error, then fix it using .clone() instead so both variables remain usable.

View solution

CHALLENGE 2

Explain why let x = 5; let y = x; println!("{} {}", x, y); compiles fine with no error, while the equivalent pattern with a String does not — referencing the Copy trait specifically.

View solution

CHALLENGE 3

Write a function that takes a String, prints it, and returns it back to the caller so the caller's variable stays usable after the call — without using .clone() anywhere.

View solution

CHAPTER 3 QUICK REFERENCE

- **Three rules:** one owner at a time; drop happens automatically at scope end; a value is never implicitly duplicated
- **Move** — assigning/passing a heap-backed value (String, Vec) transfers ownership; the original variable becomes invalid
- **.clone()** — an explicit, real deep copy; makes the cost of copying visible in the code
- **Copy trait** — simple stack-only types (i32, f64, bool, char) copy automatically on assignment; neither variable is invalidated
- **drop** — called automatically when a variable's owner goes out of scope; deterministic, no garbage collector involved
- Passing a value to a function **moves** it in, unless the type is `Copy` or the function returns it back
- "Borrow of moved value" errors are normal and common — not a sign of doing something wrong
- **Next chapter:** Borrowing & References — `&` and `&mut`, letting a function use a value without taking ownership

CHAPTER 4 OF 8

Borrowing & References

COURSE 1 · CH 4

Borrowing & References

Using a value without taking ownership of it — and completing the answer to "how does Rust replace a GC?"

Chapter 3 ended with a real frustration: passing a value to a function moves it, leaving the caller unable to use it afterward. **Borrowing** is the fix — letting a function use a value *without* taking ownership of it at all.

References With &

```
fn calculate_length(s: &String) -> usize {
    s.len()
}

fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("{} is {} bytes long", s1, len); // s1 is still valid!
}
```

`&s1` creates a **reference** to `s1` rather than moving it — `calculate_length` can read the string, but never owns it. Once the function returns, `s1` is exactly as valid as before, solving Chapter 3's function-ownership problem directly.

Mutable References With &mut

```
fn add_exclamation(s: &mut String) {
    s.push_str("!");
}

fn main() {
    let mut s = String::from("hello");
    add_exclamation(&mut s);
    println!("{}", s); // hello!
}
```

`&mut` lets a function genuinely modify the borrowed value — but this power comes with a strict rule.

The Borrow Checker's Rules

1. At any given time, you can have **either** any number of immutable references (`&`) **or** exactly one mutable reference (`&mut`) — never both at once.
2. References must always be valid — no reference to something that's already been dropped.

Both rules are enforced entirely at **compile time**.

SITUATION	ALLOWED?
Two immutable references (`&s` , `&s`) at once	Yes
One mutable reference (`&mut s`) alone	Yes
One mutable + one immutable reference, both active at once	No — compile error
Two mutable references, both active at once	No — compile error

Why These Rules Exist

The mutable-XOR-immutable rule prevents a real, classic bug class: something reading data while something else modifies it underneath it. Concretely — if two mutable references to a growing collection existed simultaneously, one holder could trigger a reallocation (the collection outgrowing its current memory and moving elsewhere) while the other still points at the *old*, now-freed memory location — a genuine dangling-pointer bug that has caused real, exploitable vulnerabilities in C++. Rust's borrow checker makes this class of bug simply **not compile**, rather than something to carefully avoid at runtime. This is also the foundation of Rust's "fearless concurrency" (Course 2) — the exact same rule that prevents this single-threaded aliasing bug also prevents data races between threads.

Dangling References

```
fn dangle() -> &String {
    let s = String::from("hello");
    &s // compile error: `s` does not live long enough
} // s is dropped here — the reference would point at freed memory
```

In C or C++, this same pattern compiles and produces a genuine dangling pointer — undefined behavior waiting to happen at runtime, possibly much later and far from where the actual mistake was made. In Rust,

it's simply a **compile error**: the borrow checker sees that `s` is dropped at the end of the function, and refuses to let a reference to it escape.

Rust's Alternative to Go's GC

The full arc from Chapter 1 is now complete. Go's garbage collector scans memory *at runtime* to find and free data nothing references anymore — genuinely useful, but with real runtime overhead and pause times that are hard to predict exactly when they'll land. Rust's borrow checker instead analyzes ownership and borrowing entirely at **compile time** — by the time a Rust program actually runs, every memory-safety guarantee has already been proven, with **zero** runtime cost for that safety, no background process ever running, and the deterministic `drop` timing Chapter 3 introduced.

RULE	WHAT IT PREVENTS
One owner at a time (Ch.3)	`Ambiguity about who's responsible for freeing memory`
Mutable XOR immutable references	`Reading data while it's being modified (aliasing bugs, data races)`
References must stay valid	`Dangling references / use-after-free`

THE COMPLETE PICTURE: CHAPTERS 1 → 4

Chapter 1 named the goal (memory safety, no GC). Chapter 3 supplied the ownership rules that make cleanup deterministic. This chapter supplies the borrow checker, which lets code actually *use* data without constantly transferring ownership back and forth — while still catching, at compile time, every aliasing bug a garbage collector was never designed to catch in the first place.

A REFERENCE'S "SCOPE" ENDS AT ITS LAST USE, NOT THE END OF THE BLOCK

Modern Rust uses **non-lexical lifetimes** — a reference is considered to have ended as soon as it's last actually used, not necessarily at the closing brace of its enclosing block. This means two references can sometimes coexist in code that looks like it should conflict, as long as the first one's last use happens before the second one is created. This is a genuine subtlety that trips up anyone relying on an older mental model (or an outdated tutorial) — Course 2's dedicated Lifetimes chapter covers the full picture; for now, just know the borrow checker is often smarter about "how long a reference lasts" than a first glance at the code might suggest.

Coding Challenges

CHALLENGE 1

Write a function count_words that takes a &String and returns the number of words (split_whitespace().count()) without taking ownership. Call it twice on the same variable to prove it remains valid after each call.

View solution

CHALLENGE 2

Write code that creates one mutable String, then attempts to hold both an immutable reference and a mutable reference to it at the same time (using both before the function ends). Run cargo run, note the exact compiler error, then fix it by ensuring the immutable reference's last use happens before the mutable one is created.

View solution

CHALLENGE 3

Explain, in your own words, why Rust's approach to memory safety has zero runtime cost compared to Go's garbage collector — referencing specifically when each language's safety mechanism actually does its work.

View solution

CHAPTER 4 QUICK REFERENCE

- **&value** — an immutable reference; the function can read but not modify, and never takes ownership
- **&mut value** — a mutable reference; allows modification, but only one may exist at a time
- **Borrow rule:** any number of `&` references, OR exactly one `&mut` reference — never both simultaneously
- **References must stay valid** — the compiler rejects any reference that could outlive the data it points to
- These rules prevent aliasing bugs and data races **at compile time** — the same foundation behind Rust's "fearless concurrency"
- **Rust vs. Go's GC:** Rust proves memory safety at compile time (zero runtime cost); Go's GC checks at runtime (real, ongoing overhead)
- Non-lexical lifetimes: a reference's effective scope ends at its **last use**, not the closing brace — full lifetime syntax is Course 2's territory
- **Next chapter:** Structs & Methods — struct definitions, impl blocks, and methods vs. associated functions

CHAPTER 5 OF 8

Structs & Methods

COURSE 1 · CH 5

Structs & Methods

Custom data types with behavior — and finally explaining what String::from has been doing all along

With ownership and borrowing established, this chapter combines fields into custom types and gives them behavior — territory Go's own `go2-1` (Structs, Methods, Pointers) already covered, making this a natural place for direct side-by-side comparison.

Defining a Struct

```
struct Rectangle {
    width: f64,
    height: f64,
}

let rect = Rectangle { width: 30.0, height: 50.0 };
println!("{}", rect.width);
```

A named collection of typed fields, accessed with dot notation — the same basic shape as a Go struct definition.

impl Blocks

Rust keeps a struct's **data** definition and its **behavior** definition separate — fields live in `struct`, methods live in a separate `impl` ("implementation") block. This is actually a genuine **similarity** to Go, not a contrast: Go also separates data (the `struct`) from behavior (a function with a receiver), unlike Java or Kotlin, which bundle both inside one class body.

```
impl Rectangle {
    // methods and associated functions go here
}
```

Methods (&self)

```
impl Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
}

let rect = Rectangle { width: 30.0, height: 50.0 };
println!("Area: {}", rect.area());
```

A **method**'s first parameter is `&self` — a borrowed reference to the instance (Chapter 4's `&` in action), giving read access to its fields without taking ownership. Called with dot syntax: `rect.area()`.

CONCEPT	GO	RUST
Where methods are defined	`Outside the struct, with a receiver`	`Inside a separate impl block`
Receiver syntax	`func (r Rectangle) Area() float64`	`fn area(&self) -> f64`
Calling a method	`rect.Area()`	`rect.area()`

Associated Functions (No self)

A function inside an `impl` block **without** `self` is an **associated function** — not a method, and it can't be called on an instance with dot syntax. It's called via `Struct::function_name()` instead. The most common use is a constructor:

```
impl Rectangle {
    fn new(width: f64, height: f64) -> Rectangle {
        Rectangle { width, height }
    }
}

let rect = Rectangle::new(30.0, 50.0);
```

This is exactly the same `::` syntax that's already appeared constantly since Chapter 1 — `String::from("hello")` is nothing more than a call to an associated function named `from` defined in `String`'s own `impl` block. Go has no real language equivalent: a Go "constructor" is just an ordinary function following a naming convention (`NewRectangle(width, height float64) Rectangle`) — Rust's `::` namespacing is a genuine language feature tied directly to the type itself, not a convention.

> SO *THAT'S* WHAT STRING::FROM WAS DOING

Every `String::from(...)` call since Chapter 1 has been calling `String`'s own `from` associated function — the exact pattern this chapter's `Rectangle::new(...)` example just defined from scratch. Nothing about it was ever special syntax; it's just a constructor-style associated function, precisely like the ones you can now write yourself.

SELF BY VALUE CONSUMES THE INSTANCE

A method can take `&self` (borrow, read-only), `&mut self` (borrow, mutable), or plain `self` (by value) — and that last form genuinely **moves** the instance into the method, per Chapter 3's move semantics. After calling a method that takes `self` by value, the original variable is no longer usable at all, exactly like passing it into any other function. Coming from a language where "this"/"self" is always just an implicit reference with no ownership implications, this is an easy trap — reach for `&self` by default, and use plain `self` only when a method is deliberately meant to consume the instance (e.g. transforming it into something else).

Coding Challenges

CHALLENGE 1

Define a struct Circle with a single field radius (f64). Write a method area(&self) that returns the circle's area (use 3.14159 for pi), and call it on an instance.

View solution

CHALLENGE 2

Add an associated function Circle::new(radius: f64) -> Circle to Challenge 1's struct, and use it to construct an instance instead of the struct-literal syntax.

View solution

CHALLENGE 3

Explain the difference between a method and an associated function in Rust, and why Struct::new(...) can't be called as instance.new(...) the way area() can be called as instance.area().

View solution

CHAPTER 5 QUICK REFERENCE

- **struct Name { field: Type, ... }** — defines a custom data type
- **impl Name { }** — a separate block holding methods and associated functions for that struct

- **Method:** first parameter is `&self` / `&mut self` / `self`; called with `instance.method()`
- **Associated function:** no `self` parameter; called with `Struct::function()` — the pattern behind every `String::from(...)` call so far
- `self` **by value** moves/consumes the instance — it's unusable afterward, per Chapter 3's move rules
- Rust and Go both separate data (struct) from behavior — unlike Java/Kotlin's class-bundled approach
- Rust's `::` constructor pattern is a real language feature; Go's `NewX()` is only a naming convention
- **Next chapter:** Enums & Pattern Matching — enums that carry data, the `match` expression, and `Option<T>` replacing null entirely

CHAPTER 6 OF 8

Enums & Pattern Matching

COURSE 1 · CH 6

Enums & Pattern Matching

Enums that actually carry data, an exhaustive match, and eliminating null as a concept entirely

Chapter 5 covered structs. This chapter covers Rust's other major custom type — and it's genuinely more powerful than what Go calls an "enum," which is really just a sequence of typed integer constants built with `iota`.

Enums That Carry Data

A Rust enum **variant** can hold its own associated data — and different variants can hold entirely different shapes of data. Go's `iota`-based constants are just named integers; there's no way to attach data to one at all.

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6(String::from("::1"));
```

`V4` and `V6` aren't just labels — each is effectively its own little data structure, holding whatever shape of data actually fits that variant.

The match Expression

Rust's `match` is a switch-like construct, but stricter and more useful: it's an **expression** (it produces a value), and it must be **exhaustive** — every possible variant needs a matching arm, or an explicit `_` catch-all. Missing a case is a **compile error**, not silent fall-through the way an incomplete `switch` in Go or JavaScript can behave.

```
fn describe(addr: &IpAddr) -> String {
    match addr {
        IpAddr::V4(a, b, c, d) => format!("IPv4: {}.{}.{}.{}", a, b, c, d),
        IpAddr::V6(s) => format!("IPv6: {}", s),
    }
}
```

Each arm **destructures** the variant's data directly — `IpAddr::V4(a, b, c, d)` pulls all four numbers out in one line, ready to use immediately.

Option<T>: No Null At All

Rust has **no null or nil value at all** — the concept doesn't exist in the language. Anything that might be absent is instead wrapped in the built-in `Option<T>` enum:

```
enum Option<T> {
    Some(T),
    None,
}
```

You cannot accidentally treat a `None` as if it were a real value — the compiler forces every `Option<T>` to be handled (via `match`, and later `if let` / `unwrap`) before the real value inside a `Some` can be used at all.

```
fn safe_divide(a: f64, b: f64) -> Option<f64> {
    if b == 0.0 {
        None
    } else {
        Some(a / b)
    }
}

match safe_divide(10.0, 2.0) {
    Some(result) => println!("Result: {}", result),
    None => println!("Cannot divide by zero"),
}
```

	GO	RUST
"Enum"	iota — just typed integer constants	Variants can each carry their own data
Missing value	nil / zero value	Option<T> — Some(T) or None
What happens if mishandled	Nil pointer dereference — a runtime panic	Compile error — must handle None before compiling

FIXING "THE BILLION DOLLAR MISTAKE"

Tony Hoare, who invented the null reference in 1965, later called it his "billion dollar mistake," estimating the cumulative cost of null-related bugs across the software industry. `Option<T>` is Rust's structural answer: since null doesn't exist as a concept at all, an entire category of bugs — null pointer/reference exceptions — simply cannot occur, caught instead as a compile-time requirement to handle absence explicitly.

A BROKEN MATCH AFTER ADDING A VARIANT IS THE COMPILER HELPING YOU

Adding a new variant to an existing enum can suddenly make every `match` elsewhere in a codebase fail to compile, if none of them have a catch-all `_` arm. This can feel like a nuisance at first — but it's the compiler doing exactly its job: pointing at every single place in the code that needs to be updated to actually handle the new case, rather than letting it silently fall through unhandled somewhere far from where the variant was added.

Coding Challenges

CHALLENGE 1

Define an enum Shape with variants Circle(f64) (radius) and Rectangle(f64, f64) (width, height). Write a match expression that computes the area for either variant.

View solution

CHALLENGE 2

Write a function `find_first_even(numbers: &[i32]) -> Option<i32>` that returns the first even number in a slice, or `None` if there isn't one. Call it with a match that prints either the found number or a "no even number" message.

View solution

CHALLENGE 3

Explain why Rust's `Option<T>` prevents a whole category of bugs that Go's nil pointers don't — specifically, at what point each language catches the mistake of using an absent value.

View solution

CHAPTER 6 QUICK REFERENCE

- **enum** variants can carry their own data — unlike Go's iota, which produces plain integer constants
- **match** is exhaustive — every variant needs an arm, or an explicit `_` catch-all, enforced at compile time

- match arms can **destructure** a variant's data directly, ready to use in the arm's body
- **Option<T>** — Rust's built-in replacement for null: `Some(T)` or `None`
- The compiler forces every `Option<T>` to be handled before the real value can be used — no null pointer exceptions possible
- Go's nil pointer dereference is a **runtime** panic; Rust's equivalent mistake is a **compile-time** error
- A match that breaks after adding a new enum variant is the compiler correctly flagging every spot that needs updating
- **Next chapter:** Error Handling — `Result<T, E>`, the `?` operator, and `panic!` vs. recoverable errors

CHAPTER 7 OF 8

Error Handling

COURSE 1 · CH 7

Error Handling

Result<T, E>, the ? operator, and knowing when a failure deserves a value instead of a crash

Chapter 6's `Option<T>` handled *absence*. This chapter handles the other "might not work" case: an operation that can fail *with a reason why* — and, genuinely more than most mainstream languages, Rust's approach here has real common ground with Go's own explicit error handling.

Result<T, E>

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Same basic shape as `Option`, but the failure case carries a **value** (`E`) explaining what went wrong, not just an absence.

```
fn divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err(String::from("cannot divide by zero"))
    } else {
        Ok(a / b)
    }
}

match divide(10.0, 0.0) {
    Ok(result) => println!("Result: {}", result),
    Err(message) => println!("Error: {}", message),
}
```

Rust making errors part of the function's *return type*, rather than an exception, is genuinely the same core philosophy Go's own `(value, error)` convention already uses — a real similarity, not just a contrast, distinguishing both languages from JavaScript/Python's try/catch exception model.

panic! vs. Recoverable Errors

`panic!` immediately stops the program — reserved for genuinely **unrecoverable** situations: a real bug, a broken invariant, something that should structurally never happen. `Result` is for **expected, recoverable** failure conditions: a missing file, invalid user input, a failed network call. Rust's convention is clear about which to reach for: `Result` for anything a caller might reasonably want to handle; `panic!` only for "this should never happen."

The ? Operator

Go's error handling requires an explicit check after *every single call* that can fail: `if err != nil { return err }`. Rust's `?` operator does the exact same propagation in one character: if the expression is `Err`, it returns that `Err` immediately from the current function; if it's `Ok`, it unwraps the value and execution continues.

```
fn calculate(a: f64, b: f64, c: f64) -> Result<f64, String> {
    let step1 = divide(a, b)?;   // propagates Err immediately if this fails
    let step2 = divide(step1, c)?;
    Ok(step2)
}
```

	GO	RUST
Error type	A separate error return value	Err(E) variant of Result<T, E>
Propagating an error up	if err != nil { return err } — repeated at every call	? — one character, same call site

unwrap() and expect()

`.unwrap()` extracts the `Ok` / `Some` value directly — or **panics** if it's actually `Err` / `None`. `.expect("message")` does the same, with a custom panic message. Both are genuinely useful for quick prototypes or cases that are truly certain to succeed — but reaching for them on an operation that could realistically fail in production turns a recoverable situation into a full program crash.

> **GO AND RUST ACTUALLY AGREE HERE**
>
> Of the languages compared throughout this course, Go and Rust share the most genuine common ground on error handling: both make errors an explicit part of a function's signature/return value, forcing the caller to acknowledge them, rather than letting them propagate invisibly as exceptions the way JavaScript, Python, and Java's try/catch model does. Rust's `?` is best understood as a more concise expression of the exact same idea Go's repeated `if err != nil` already embodies — not a fundamentally different philosophy.

.UNWRAP() PANICKING IN PRODUCTION IS A REAL, COMMON MISTAKE

It's tempting to reach for `.unwrap()` everywhere during early development, since it's shorter than a full `match`. The real risk: any call that genuinely *can* fail in production (parsing user input, a network request, a file read) will crash the entire program the moment it does, instead of being handled gracefully. Reserve bare `.unwrap()` for prototypes, tests, or situations that are truly structurally impossible to fail — everywhere else, use `match`, `?`, or at minimum `.expect("a clear message")` so a failure at least explains itself before the crash.

Coding Challenges

CHALLENGE 1

Write a function `parse_age(input: &str) -> Result<u8, String>` that attempts to parse a string into a u8, returning a clear error message on failure. Call it with both a valid and an invalid input, handling both with match.

View solution

CHALLENGE 2

Rewrite this chapter's calculate function (which chains two divide calls using ?) as if Rust had no ? operator at all — using explicit match statements for each call instead. Compare the length and readability to the ? version.

View solution

CHALLENGE 3

Explain when panic! is the right choice versus Result, using one concrete example of each — and explain what's risky about calling .unwrap() on a Result that comes from parsing user-provided input.

View solution

CHAPTER 7 QUICK REFERENCE

- **Result<T, E>** — Ok(T) for success, Err(E) for failure with an explanatory value
- **panic!** — for unrecoverable bugs/invariant violations, stops the program immediately
- **Result** — for expected, recoverable failures (bad input, a missing file, a failed request)
- **?** — propagates an Err immediately, or unwraps Ok and continues; the concise version of Go's repeated `if err != nil`
- **.unwrap() / .expect("msg")** — extract the value or panic; fine for prototypes/tests, risky on anything that can genuinely fail in production

- Go and Rust share a real philosophy: errors as explicit return values, not exceptions
- **Next chapter:** Collections & Iterators — Vec<T>, HashMap<K,V>, iterator adapters, and ownership implications of iterating

CHAPTER 8 OF 8

Collections & Iterators

COURSE 1 · CH 8

Collections & Iterators

Vec<T>, HashMap<K,V>, and the functional-style iterator chains that process them

Chapter 2's arrays were fixed-size. This final chapter of Course 1 covers Rust's **growable** collections — and the iterator adapters that process them in a style that will feel immediately familiar coming from JavaScript.

Vec<T>: A Growable Array

```
let mut scores: Vec<i32> = Vec::new();
scores.push(90);
scores.push(85);

let scores2 = vec![90, 85, 78]; // shorthand macro for creating one with initial values

for score in &scores2 {
    println!("{}", score);
}
```

`Vec<T>` can grow and shrink at runtime, backed by a heap allocation — genuinely similar to Go's own **slice** type, which is Go's own answer to "an array that can grow." This is a real similarity worth noting, not just another contrast.

HashMap<K, V>: Key-Value Storage

```
use std::collections::HashMap;

let mut ages: HashMap<String, i32> = HashMap::new();
ages.insert(String::from("Dana"), 30);

match ages.get("Dana") {
    Some(age) => println!("Dana is {}", age),
    None => println!("Not found"),
}
```

`get` returns `Option<&V>` — Chapter 6's `Option` making a direct return appearance. Go's built-in `map[string]int` is more tightly woven into the language's own syntax, and its lookup returns a `(value, ok bool)` pair — both languages are explicit about "this key might not exist," just expressed through different mechanisms (Go's two-value return vs. Rust's `Option`).

Iterator Adapters: map, filter, collect

Rust's iterator methods will feel immediately familiar coming from JavaScript's own array methods — the naming and behavior are genuinely close.

```
let numbers = vec![1, 2, 3, 4, 5];

let doubled_evens: Vec<i32> = numbers
    .iter()
    .filter(|&&n| n % 2 == 0)
    .map(|&n| n * 2)
    .collect();

println!("{:?}", doubled_evens); // [4, 8]
```

`.iter()` creates an iterator of **references** (borrowing, Chapter 4) without consuming `numbers`. `.filter()` and `.map()` are **lazy** — nothing runs until `.collect()` actually consumes the chain and builds a new `Vec`.

	JAVASCRIPT	RUST
Transform each element	`array.map(x => ...)`	`.iter().map(|x| ...)`
Keep matching elements	`array.filter(x => ...)`	`.iter().filter(|x| ...)`
Execution	`Eager — runs immediately`	`Lazy — runs only when collected/consumed`

Ownership Implications of Iterating

Rust has three distinct ways to iterate, and choosing the wrong one is a genuinely common point of confusion:

METHOD	YIELDS	COLLECTION USABLE AFTER?
`.iter()`	`&T (borrowed references)`	`Yes`
`.iter_mut()`	`&mut T (mutable references)`	`Yes`
`.into_iter()` / bare `for x in vec`	`T (owned values)`	`No — consumed/moved (Ch.3)`

Neither Go nor JavaScript has this distinction at all — their iteration never "consumes" the underlying collection the way `into_iter()` does.

FULL CIRCLE: COURSE 1'S ARC, ONE MORE TIME

`Vec<T>` and `HashMap<K,V>` are both heap-allocated — meaning Chapter 3's move semantics apply to them exactly as they did to `String`. Assigning one `Vec` to another variable moves it, not copies it; `.clone()` is still the explicit way to get a real independent duplicate. Every chapter in this course has been building toward being able to read code like this one's `filter` / `map` / `collect` chain and immediately know exactly what's borrowed, what's owned, and what's been moved.

"TYPE ANNOTATIONS NEEDED" IS COLLECT()'S MOST COMMON COMPLAINT

`collect()` is generic over what it builds — it needs to know the target type from somewhere, either an explicit variable type annotation (`let x: Vec<i32> = ...`) or the "turbofish" syntax (`.collect::<Vec<i32>>()`). Omitting both produces a real, common compiler error demanding a type annotation — not a bug in the code's logic, just `collect()` genuinely being unable to infer what to build without more information.

Coding Challenges

CHALLENGE 1

Create a Vec of five integers. Using an iterator chain (.iter(), .filter(), .map(), .collect()), produce a new Vec containing only the odd numbers, each squared.

View solution

CHALLENGE 2

Create a HashMap mapping three product names (String) to their prices (f64). Look up one product that exists and one that doesn't, handling both cases with match on the Option returned by get.

View solution

CHALLENGE 3

Explain the difference between iterating with .iter(), .iter_mut(), and into_iter() (or a bare for x in vec), specifically in terms of what happens to the original collection afterward in each case.

View solution

CHAPTER 8 QUICK REFERENCE

- **Vec<T>** — a growable, heap-backed array; `vec![...]` macro or `Vec::new()` + `.push()`
- **HashMap<K, V>** — key-value storage; `.get()` returns `Option<&V>`
- **.iter().map(...).filter(...).collect()** — a lazy, chainable iterator pipeline, close to JavaScript's array methods
- **.iter()** / **.iter_mut()** — borrow (immutably/mutably); the collection remains usable afterward
- **.into_iter()** / bare `for x in vec` — consumes the collection; it's moved and unusable afterward
- **collect()** needs a target type — via variable annotation or the turbofish `::<Type>()`
- Vec and HashMap are heap-allocated — Chapter 3's move semantics apply to both directly

★ Rust Fundamentals Complete — 8 / 8 chapters

From cargo and the toolchain through variables, ownership, borrowing, structs, enums, error handling, and finally collections and iterators — you now have the complete foundation Rust's whole design is built on. Next up: **Rust Intermediate**, covering lifetimes, traits, generics, smart pointers, concurrency, modules, and testing.