What is Cargo?

In PHP

Composer starts a project, reads `composer.json`, writes `composer.lock`, and downloads packages from Packagist. You still run PHP yourself. Cargo is that, and it is also the thing that compiles, tests, formats, and lints. One tool, the way you wish `php` and `composer` were sometimes the same command.

In Rust

`cargo new` creates a directory, a git repo, `Cargo.toml`, and `src/main.rs`. `cargo init` does the same in the current directory. Libraries (`cargo new --lib`) are in the modules chapter.

a directory you can write to

```
cargo new hello-cargo
cd hello-cargo
```

The generated program is small.

src/main.rs

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

Cargo.toml

```
[package]
name = "hello-cargo"
version = "0.1.0"
edition = "2024"
```

`edition = "2024"` is the language edition. It is not the compiler version. Leave it on 2024 for this course.

crate root

```
 cargo run
cargo fmt
cargo clippy
cargo build
cargo build --release
```

- `cargo run` compiles and runs `src/main.rs`.
- `cargo fmt` formats the crate. Run it before you commit.
- `cargo clippy` is the linter. Default lints are enough for this course. In a crate where you want stricter advice, you can add `#![warn(clippy::pedantic)]` at the top of `src/main.rs`; this course does not require a clean pedantic run.
- `cargo build` writes `target/debug/hello-cargo`. `cargo build --release` writes `target/release/hello-cargo`, optimised. Release is faster; debug keeps more information for backtraces. Debug builds also **panic on integer overflow**; release builds wrap. That difference is why the calculator later uses `checked_*` instead of plain `+` and `/`.

Keep warnings as warnings while you work locally. In CI, treat them as errors with `RUSTFLAGS="-D warnings"` before you build, or run `cargo clippy -- -D warnings`. An unused `Result` then fails the pipeline (`unused_must_use`) instead of scrolling past. This course points at that CI setup from here on. Do not put `warnings = "deny"` under `[lints.rust]` in `Cargo.toml` for day-to-day work: every unused variable, field, or import becomes a hard error, which fights incomplete code.

`cargo check` type-checks without writing the executable. It is the fast loop while you are fixing errors.

> **Watch for:** hand-editing a version into `Cargo.toml` for every crate. From the next lesson, `cargo add` does that.

> **Idiomatic Rust:** commit the source, not `target/`. Cargo recreates `target/` when you build.

Side by side

	PHP	Rust
New project	`composer init` (or `composer create-project` from a template)	`cargo new`
Build and run	You run `php` yourself	`cargo build` and `cargo run`
Format and lint	Extra tools	`cargo fmt` and `cargo clippy`

Try it

`cargo new` a project, change the string in `println!`, and `cargo run`. Then `cargo fmt` and `cargo clippy`. Clippy should print nothing.

Key takeaways

- `cargo new` makes a directory. `cargo init` uses the one you are in.
- `cargo run`, `cargo fmt`, and `cargo clippy` are the everyday commands.
- `cargo check` is the fast loop while fixing errors.

What's next: Managing dependencies with Cargo.toml