TD-6

Programming methodology: crates, modules and third-party libraries

Pierre Cochard and Tanguy Risset

5TC-RUST 2025

Summary

- Packages and crates
- Modules: declaration, visibility, paths, files
- Tests, examples and documentation
- Using third-party libraries: finding, choosing, adding crates
- Versions, `Cargo.lock` and features
- Workspaces and code quality tools

Why this course?

- Real projects are split into **several files and modules**, each with a clear responsibility.
- Part of the code is often meant to be **reused**, as a **library**.
- The Rust standard library is deliberately **small**: no random numbers, no JSON, no command line parser, no HTTP… and **no** `async` **runtime** (next course!).
- These come from third-party libraries (***crates***) published on crates.io.

Packages and crates

- A **crate** is the *compilation unit* of Rust, a tree of modules starting from a **crate root**:
  - **binary crate**: a program, with a `main()` function, root `src/main.rs`;
  - **library crate**: no `main()`, code meant to be **shared**, root `src/lib.rs`.
- A **package** (created by `cargo new`) is a directory with a `Cargo.toml`, which contains:
  - **at most one** library crate;
  - **any number** of binary crates.

Cargo conventions

```
my_package/
├── Cargo.toml          # package description and dependencies
├── Cargo.lock          # exact versions of all dependencies (generated)
├── src/
│   ├── lib.rs          # root of the library crate (optional)
│   ├── main.rs         # root of the default binary crate (optional)
│   └── bin/tool1.rs    # additional binary crate 'tool1'
├── tests/              # integration tests
├── examples/           # examples (cargo run --example name)
└── target/             # build artifacts (never commit this!)
```

```
cargo new my_app            # binary crate (src/main.rs)
cargo new --lib my_lib      # library crate (src/lib.rs)
cargo run --bin tool1       # runs src/bin/tool1.rs
```

Question 1)

- Create a library package with `cargo new --lib geometry`.
- Look at the generated `src/lib.rs` file: what does it contain?
- Run `cargo build`, then `cargo run`: what happens, and why?
- Finally, run `cargo test`.

Binary and library in the same package

- Recommended layout: **both** `src/lib.rs` and `src/main.rs`:
  - `src/lib.rs` contains **all the logic**: easy to test and to reuse;
  - `src/main.rs` stays **small**: parses arguments, calls the library, displays results.
- The binary uses the library **through its package name**, and only sees its **public** items:

```
// src/lib.rs (package name: 'geometry')
pub fn area(width: f64, height: f64) -> f64 { width * height }

// src/main.rs
fn main() {
    println!("area = {}", geometry::area(2.0, 3.0));
}
```

Question 2)

- In your `geometry` package, add a `src/main.rs` file.
- In `src/lib.rs`, remove the generated code and write a function `rectangle_area(w: f64, h: f64) -> f64`.
- Call it from `main()`.
- What happens if you remove the `pub` keyword from the function?

Modules

- Inside a crate, code is organized in **modules**: namespaces that group related items and control their **visibility**.

```
mod shapes {
    pub mod circle {
        pub fn area(r: f64) -> f64 { std::f64::consts::PI * r * r }
    }
    pub mod rectangle {
        pub fn area(w: f64, h: f64) -> f64 { w * h }
    }
}

fn main() {
    println!("{}", shapes::circle::area(1.0));
    println!("{}", shapes::rectangle::area(2.0, 3.0));
}
```

- You already used them: `std::collections::HashMap`.

Visibility and privacy

- **Everything is private by default**: visible in its module and its children only.

Modifier	Visible from
(nothing)	the current module and its children
`pub(super)`	the parent module
`pub(crate)`	the whole current crate
`pub`	everywhere, including other crates

- `struct` fields have their **own visibility**: a `pub struct` can have private fields (**encapsulation**).
- If an `enum` is `pub`, all its variants are public.

Encapsulation example

```
mod bank {
    pub struct Account {
        pub owner: String,
        balance: i64, // private: cannot be modified from outside 'bank'
    }
    impl Account {
        pub fn new(owner: &str) -> Self {
            Account { owner: owner.to_string(), balance: 0 }
        }
        pub fn deposit(&mut self, amount: i64) {
            assert!(amount > 0);
            self.balance += amount;
        }
        pub fn balance(&self) -> i64 { self.balance }
    }
}
```

Question 3)

- Copy the `bank` example. Try to create an `Account` directly from `main()` with the `Account { owner: ..., balance: ... }` syntax. What error do you get?
- Add a `withdraw(&mut self, amount: i64) -> Result<(), String>` method that refuses to make the balance negative.
- Why is the `balance` field being private important here?

Paths and the `use` keyword

- **Absolute paths** start from the crate root: `crate::...` (or an external crate: `std::...`).
- **Relative paths** start from the current module: `self::...`, `super::...` (parent).
- `use` creates a **shortcut** to a path:

```
use std::collections::HashMap;          // a single item
use std::io::{self, Read, Write};       // several items
use std::fmt::Result as FmtResult;      // renaming
```

- Convention: import the **parent module** of functions, the **item itself** for types and traits.
- `pub use` **re-exports** an item: a simple public API, independent of the internal organization.

Question 4)

Replace each `todo!()` with a correct path (try both `crate::` and `super::`), then simplify `main()` with `use`:

```
mod network {
    pub fn connect() -> &'static str { "connected" }
    pub mod server {
        pub fn start() -> String {
            let c = todo!(); // call 'network::connect()' here
            format!("server started, {c}")
        }
    }
}
mod client {
    pub fn run() -> String {
        todo!() // call 'network::server::start()' here
    }
}
fn main() {
    println!("{}", client::run());
    println!("{}", network::server::start());
}
```

Splitting modules into files

- `mod name { ... }` defines a module **inline**; `mod name;` loads it **from a file**: `src/name.rs` (or `src/name/mod.rs`).
- A child module `parent::child` is declared **in the parent's file**, and lives in `src/parent/child.rs`.

```
src/
├── main.rs           // mod shapes;
├── shapes.rs         // pub mod circle; pub mod rectangle;
└── shapes/
    ├── circle.rs     // pub fn area(r: f64) -> f64 { ... }
    └── rectangle.rs
```

- `mod` is **not** an `#include`: a file is compiled **only if** it is declared with `mod`, exactly **once**.
- The file *is* the module: no `mod name { }` around its content.

Questions 5) and 6)

- **Question 5**: Create a package `shapes`. In a **single** `src/main.rs`, write a module `shapes` with two sub-modules `circle` and `rectangle`, each defining a `pub struct` (`Circle`, `Rectangle`) with an `area(&self) -> f64` method. Then **move** each module to its own file, without changing `main()`.
- **Question 6**: Add a `Shape` trait with an `area(&self) -> f64` method, implemented by both types. Where should it be defined? Re-export `Circle`, `Rectangle` and `Shape` from `shapes` with `pub use`, and write `total_area(shapes: &[Box<dyn Shape>]) -> f64`.

Tests, examples and documentation

- **Unit tests**: in the **same file**, in a `#[cfg(test)] mod tests`, which can access **private** items:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn unit_circle_area() {
        assert!((Circle { r: 1.0 }.area() - std::f64::consts::PI).abs() < 1e-12);
    }
}
```

- **Integration tests**: in `tests/`, each file is a **separate crate** that only sees the **public API** of `src/lib.rs`.
- `cargo test` runs unit tests, integration tests and doc-tests.

Documenting your crate

- `///` documents the **next item**, `//!` the **enclosing** item (e.g. the crate, at the top of `lib.rs`).
- Written in Markdown; code examples are compiled and run by `cargo test` (**doc-tests**).

```rust
/// Computes the area of a rectangle.
///
/// # Examples
/// ```
/// assert_eq!(geometry::rectangle_area(2.0, 3.0), 6.0);
/// ```
pub fn rectangle_area(w: f64, h: f64) -> f64 { w * h }
```

- `cargo doc --open`: documentation of your crate **and all its dependencies** (same as docs.rs).

Question 7)

- In your `geometry` package, add:
  - a unit test,
  - an integration test in `tests/`,
  - a documented function with an example.
- Run `cargo test` and identify the three sections of the output.
- Run `cargo doc --open` and look at the result.

Using third-party libraries

Where to search for crates:

- **crates.io**: the official registry, search by keyword or **category**, sort by downloads.
- **lib.rs**: an alternative index, with better categories and rankings.
- **blessed.rs**: a curated list of the *de facto* standard crates. A very good start!
- **docs.rs**: the documentation of **every version** of **every** crate.
- From the command line: `cargo search <keyword>`.

Crates you should know

Need	Crate(s)
Random numbers	`rand`
Command line arguments	`clap`
(De)serialization (JSON, TOML, …)	`serde`, `serde_json`, `toml`
Error handling	`anyhow` (applications), `thiserror` (libraries)
Logging	`log`, `env_logger`, `tracing`
Regular expressions	`regex`
Date and time	`chrono`, `jiff`
Async runtime, networking	`tokio` (next course!), `reqwest`
Data parallelism	`rayon`
Benchmarking	`criterion`, `divan`

Choosing a crate

A dependency is code you **trust**: it is compiled into your program, and can run code at build time. Check:

1. **Popularity**: downloads, number of *reverse dependencies*.
2. **Maintenance**: date of the last release, repository activity.
3. **Maturity**: `1.x` = stable API; `0.x` = expect breaking changes.
4. **Documentation**: README, `examples/`, docs.rs page.
5. **License**: usually `MIT OR Apache-2.0`.
6. **Dependencies**: fewer is better (compile time, attack surface).
7. **Security**: RustSec database, `cargo audit`.

Question 8)

- You need to generate random numbers in a program.
- Find the most commonly used crate for this purpose.
- On its crates.io page, find: its latest version, its license, its number of downloads, and the links to its documentation and repository.
- On docs.rs, find a function that generates a random integer within a given range.

Adding a dependency

```
cargo add rand                          # latest version from crates.io
cargo add serde --features derive       # with an optional feature
cargo add rand@0.9                      # a specific version
cargo add --dev criterion               # only for tests/benchmarks/examples
cargo remove rand
```

```toml
[dependencies]
rand = "0.10"
serde = { version = "1.0", features = ["derive"] }
my_local_lib = { path = "../my_local_lib" }
```

- The crate is then directly usable by its name: `rand::random_range(1..=6)`.
- `cargo tree`: dependency tree. `cargo doc --open`: docs of the **exact versions** you use.

Question 9)

- Create a new package `guess` and add the `rand` crate.
- Write a "*guess the number*" game: the program picks a random number between 1 and 100, then reads numbers from the standard input (`std::io::stdin().read_line(...)`) and prints "too small", "too big" or "found!" until the user finds it.
- Run `cargo tree`: how many crates are actually compiled?

Versions and `Cargo.lock`

- Crates follow **Semantic Versioning**: `MAJOR.MINOR.PATCH` (for `0.x.y`, `x` acts as the major number).
- `rand = "0.10.1"` means "*any version compatible with 0.10.1*": `>=0.10.1, <0.11.0`.
- The **exact** versions used are recorded in **`Cargo.lock`**: reproducible builds. **Always commit it.**
- `cargo update`: latest *compatible* versions. A new **major** version must be chosen explicitly in `Cargo.toml`.
- Two major versions of the same crate are **different crates** (`cargo tree --duplicates`).

Question 10)

- Open the `Cargo.lock` file of your `guess` package and find the entry for `rand`. What information does it contain?
- Change the version requirement of `rand` to `"0.8"` in `Cargo.toml` and run `cargo build`: what happens?
- What do you need to change to make it compile again? (use docs.rs)
- Restore the latest version afterwards.

Features

- Optional functionalities of a crate, listed in its "*Feature flags*" docs.rs menu:

```
serde = { version = "1.0", features = ["derive"] }
tokio = { version = "1", features = ["full"] }
```

```
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug)]
struct Student { name: String, grades: Vec<f32> }

let json = serde_json::to_string_pretty(&s)?;
let back: Student = serde_json::from_str(&json)?;
```

- `cannot find derive macro 'Serialize'` or `could not find 'main' in 'tokio'`: **a feature is missing**.

Question 11)

- In a new package, add `serde` and `serde_json` **without** any feature, and write the `Student` example. What error do you get? Fix it.
- Write a `Vec<Student>` to a file `students.json` (`std::fs::write`).
- Write a second program (`src/bin/read.rs`) that reads it back and prints the average grade of each student.
- Hint: where should the `Student` type be defined to be shared by both programs?

Workspaces

- Several **packages** sharing the same `Cargo.lock` and `target/` directory:

```
# my_project/Cargo.toml
[workspace]
resolver = "3"
members = ["core", "cli", "server"]

# my_project/cli/Cargo.toml
[dependencies]
core = { path = "../core" }
```

- `cargo build` builds all members, `cargo run -p cli` runs one of them.
- For your project, a single package with `lib.rs` + `main.rs` is usually enough to begin with.

Code quality tools

- **`cargo fmt`**: formats your code with the official style.
- **`cargo clippy`**: a *linter* with hundreds of checks (mistakes, non-idiomatic code, performance).
- **`cargo check`**: type-checks without building, fast (used by `rust-analyzer`).
- Extensions: `cargo install cargo-audit`, `cargo-outdated`, `bacon`, `cargo-expand`...

Question 12)

- Run `cargo clippy` on one of the programs you wrote in a previous course. Read the warnings and fix them.

Question 13): putting it all together

Write a `wordstats` tool: `cargo run -- --top 5 --min-len 4 some_text.txt` displays the number of lines, words and characters, and the `top` most frequent words of at least `min-len` characters.

- `src/main.rs` : **only** parses arguments (with `clap` and its `derive` feature) and displays the results;
- `src/lib.rs` : declares the modules and re-exports the public API;
- `src/stats.rs` : `pub struct Stats` and `pub fn compute(text: &str) -> Stats` ;
- `src/frequency.rs` : `pub fn top_words(text: &str, n: usize, min_len: usize) -> Vec<(String, usize)>` ;
- unit tests, and an integration test in `tests/` ;
- errors propagated with `?` up to `main()` ; check with `cargo clippy` and `cargo fmt` .

Question 14): preparing the next course

- In a new package, run `cargo add tokio`, and try to compile:

```rust
#[tokio::main]
async fn main() {
    tokio::time::sleep(std::time::Duration::from_secs(1)).await;
    println!("Hello from tokio!");
}
```

- Using the documentation of `tokio` on docs.rs, find out which **features** must be enabled to make it compile.

End of TD6