Introductory Rust (2026): Structs, Modules, Errors

Lecture 3

Lukáš Hozda

Winter 2026/27

MFF CUNI

Structuring Code

Struct Design

```
struct User {
    login: String,
    quota: u64,
}
```

- Group related fields.
- Keep invariants near constructors.

Impl Blocks

```
impl User {
    fn new(login: String, quota: u64) -> Self {
        Self { login, quota }
    }

    fn consume(&mut self, amount: u64) {
        self.quota = self.quota.saturating_sub(amount);
    }
}
```

Module Boundaries

- `mod` organizes code into files.
- `pub` controls visibility.
- Expose minimal public surface.

`use` and Paths

- Absolute: `crate::module::Type`.
- Relative paths inside module.
- Re-export with `pub use` when beneficial.

Error Enum Pattern

```
#[derive(Debug)]
enum ParseError {
    MissingField,
    InvalidNumber,
}
```

- Domain errors are clearer than strings.

Result-Based Constructors

```rust
fn parse_user(line: &str) -> Result<User, ParseError> {
    let (name, quota) = line.split_once(':').ok_or(ParseError::MissingField)?;
    let quota = quota.parse::<u64>().map_err(|_| ParseError::InvalidNumber)?;
    Ok(User::new(name.to_string(), quota))
}
```

API Cohesion

- Keep parsing logic with domain type.
- Keep formatting predictable for tests.
- Keep mutable state transitions explicit.

Testability

- Small pure functions are easiest to test.
- Separate parsing, state update, and I/O.

Practical Rule

- Design module API first.
- Hide implementation details unless needed.