Introductory Rust (2026): Enums and Pattern Matching

Lecture 2

Lukáš Hozda

Winter 2026/27

MFF CUNI

Modeling States

Algebraic Data Types In Rust

- `enum` expresses closed set of variants.
- Variants can carry data.
- Pattern matching forces explicit case handling.

Enum Example

```
enum Command {
    Add(i64, i64),
    Sub(i64, i64),
    Print(String),
}
```

Exhaustive Match

```
fn eval(cmd: Command) {
    match cmd {
        Command::Add(a, b) => println!("{}", a + b),
        Command::Sub(a, b) => println!("{}", a - b),
        Command::Print(s) => println!("{s}"),
    }
}
```

- Compiler checks all variants handled.

Option<T>

```
fn parse_num(s: &str) -> Option<i64> {
    s.parse().ok()
}
```

- `Some(value)` / `None` for nullable semantics.
- Forces caller to handle missing values.

Result<T, E>

```
fn read_count(s: &str) -> Result<usize, std::num::ParseIntError> {
    s.parse::<usize>()
}
```

- Success and error paths are explicit.

`if let` and `while let`

```
if let Some(v) = parse_num("42") {
    println!("{v}");
}
```

- Good for single-variant focus.

Pattern Features

- Destructuring tuples and structs.
- Guards: `if cond` in match arm.
- _ wildcard for intentionally ignored data.

Domain Modeling

- Prefer domain enums over bool flags.
- Fewer impossible states.
- Clearer transition logic.

Error Types

- Small enums for predictable errors.
- Conversions via `From` where appropriate.
- Human-readable messages in CLI output.

Practical Rule

- Use enums when state set is finite and important.
- Let exhaustive matching guide correct behavior.