Introductory Rust (2026): Async Foundations with Tokio

Lecture 6

Lukáš Hozda

Winter 2026/27

MFF CUNI

Async Fundamentals

Why Async

- Large concurrency with limited threads.
- Good for I/O-heavy workloads.
- Not automatically faster for CPU-bound tasks.

`async fn` and `await`

```
async fn compute() -> i64 {
    42
}
```

- `async fn` returns a future.
- Work progresses when polled by runtime.

Tokio Entry Point

```
#[tokio::main]
async fn main() {
    let v = compute().await;
    println!("{v}");
}
```

Spawning Tasks

```
let h1 = tokio::spawn(async { 10 });
let h2 = tokio::spawn(async { 20 });
println!("{}", h1.await.unwrap() + h2.await.unwrap());
```

Cancellation Basics

- Dropping `JoinHandle` detaches task.
- Explicit cancellation requires design.
- Keep cleanup idempotent.

Timeouts

```
use tokio::time::{timeout, Duration};

let result = timeout(Duration::from_millis(50), async { 7 }).await;
println!("{:?}", result);
```

Async Channels

- Tokio channels for task coordination.
- Backpressure is part of correctness model.

Common Pitfalls

- Blocking call inside async task.
- Forgetting `.await` and dropping future.
- Shared mutable state without synchronization.

Practical Rule

- Keep async boundaries explicit.
- Separate CPU-heavy work from async executor threads.