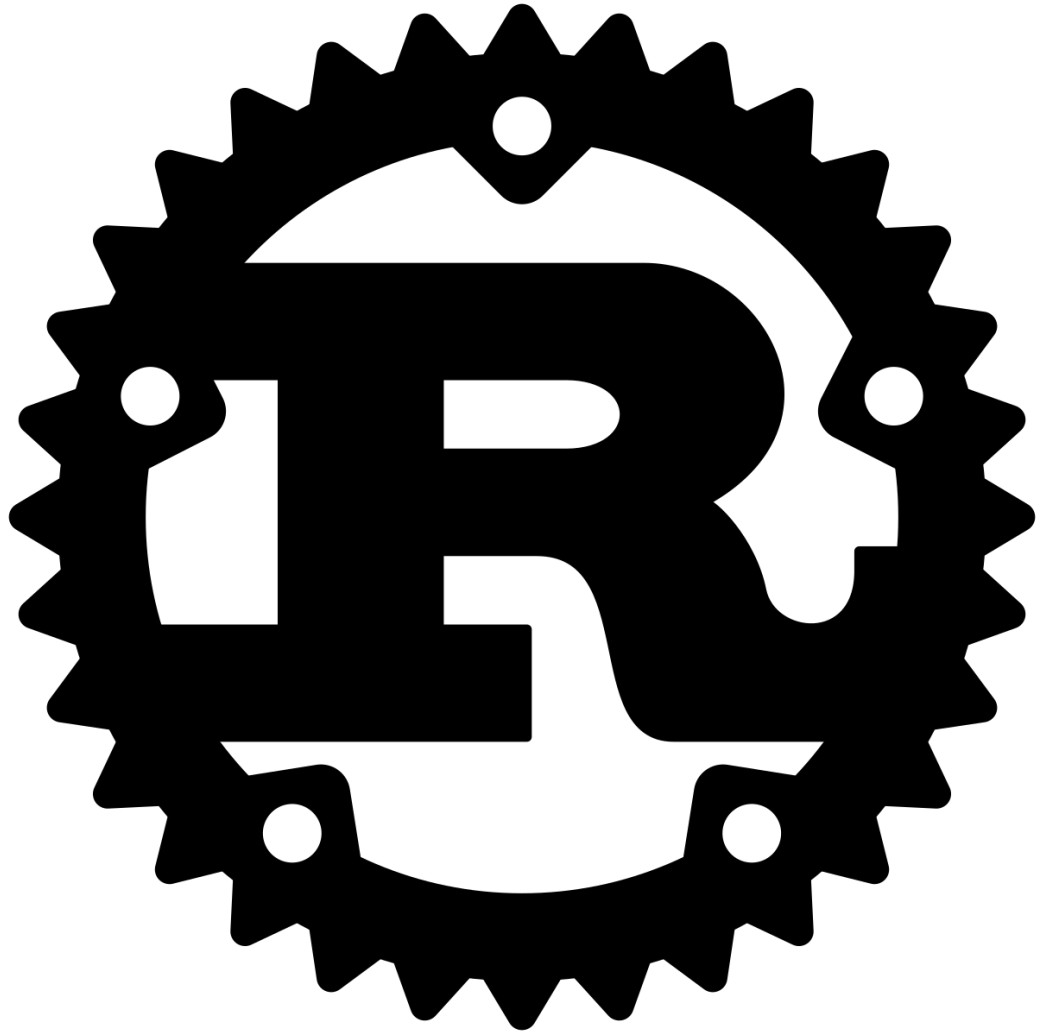
Rust

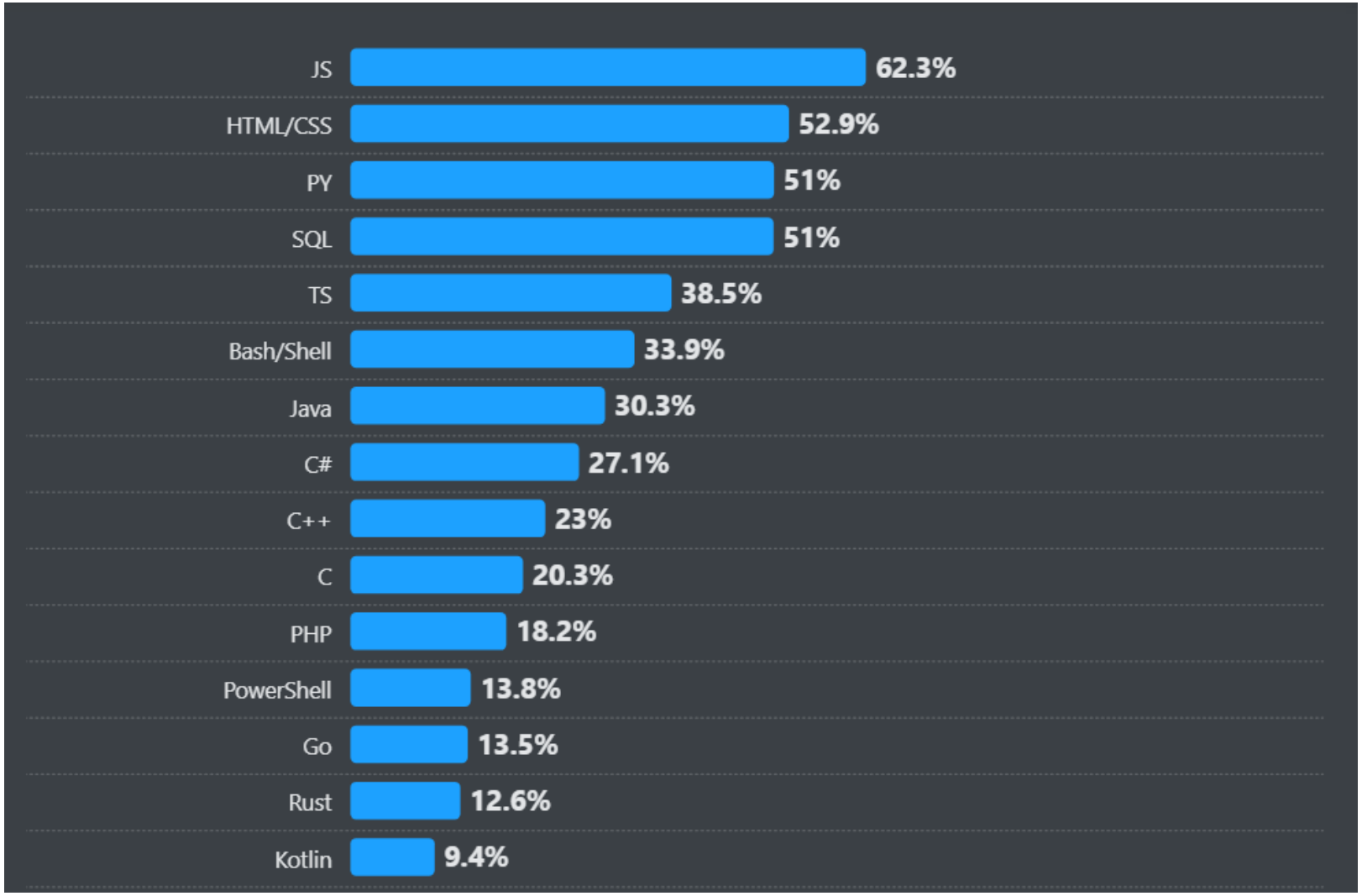
PCP – Hochschule Luzern



Vision/Geschichte

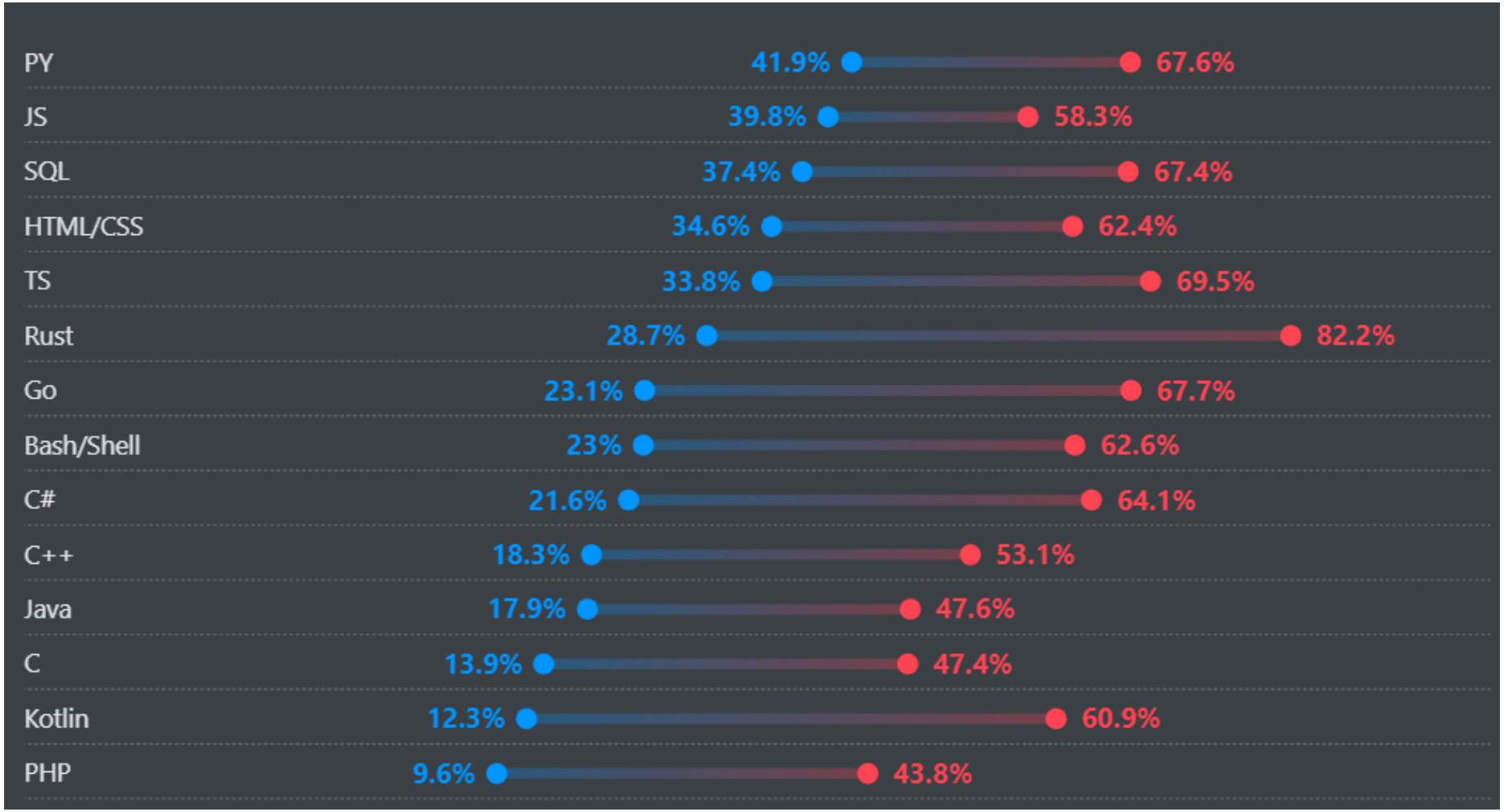
- Mozilla Research
- Erste Stabile Version 2015
- Alternative zu C und C++
- Sicher, Schnell und Modern

Verbreitung



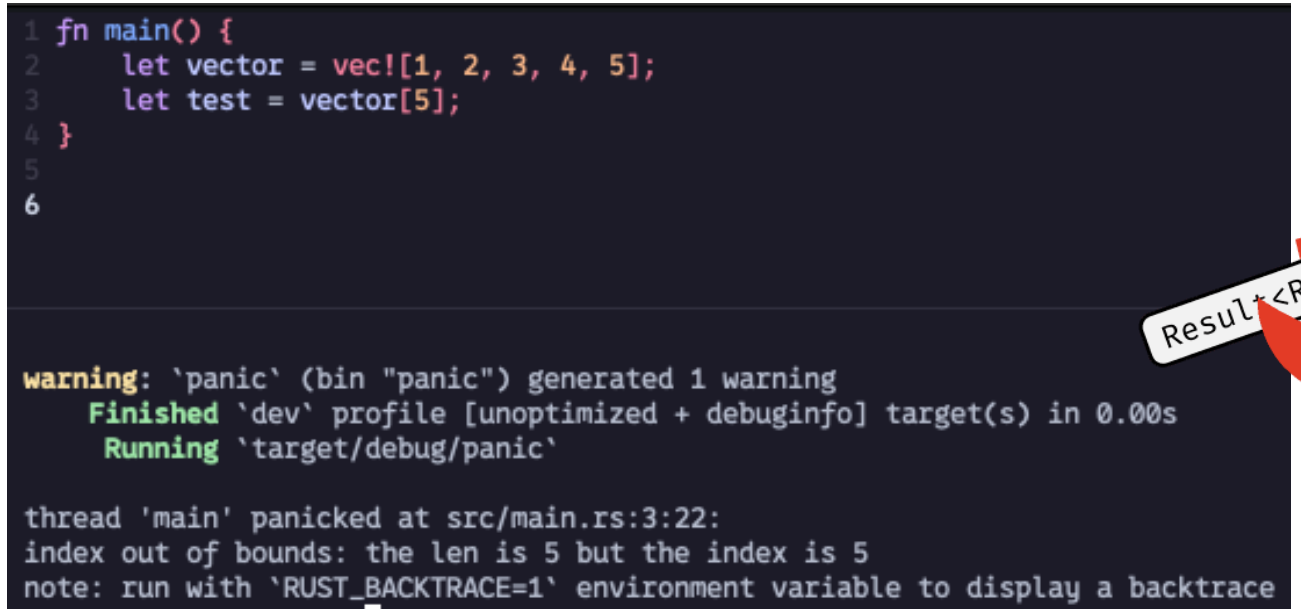
https://survey.stackoverflow.co/2024

Aber!



https://survey.stackoverflow.co/2024

Panic!

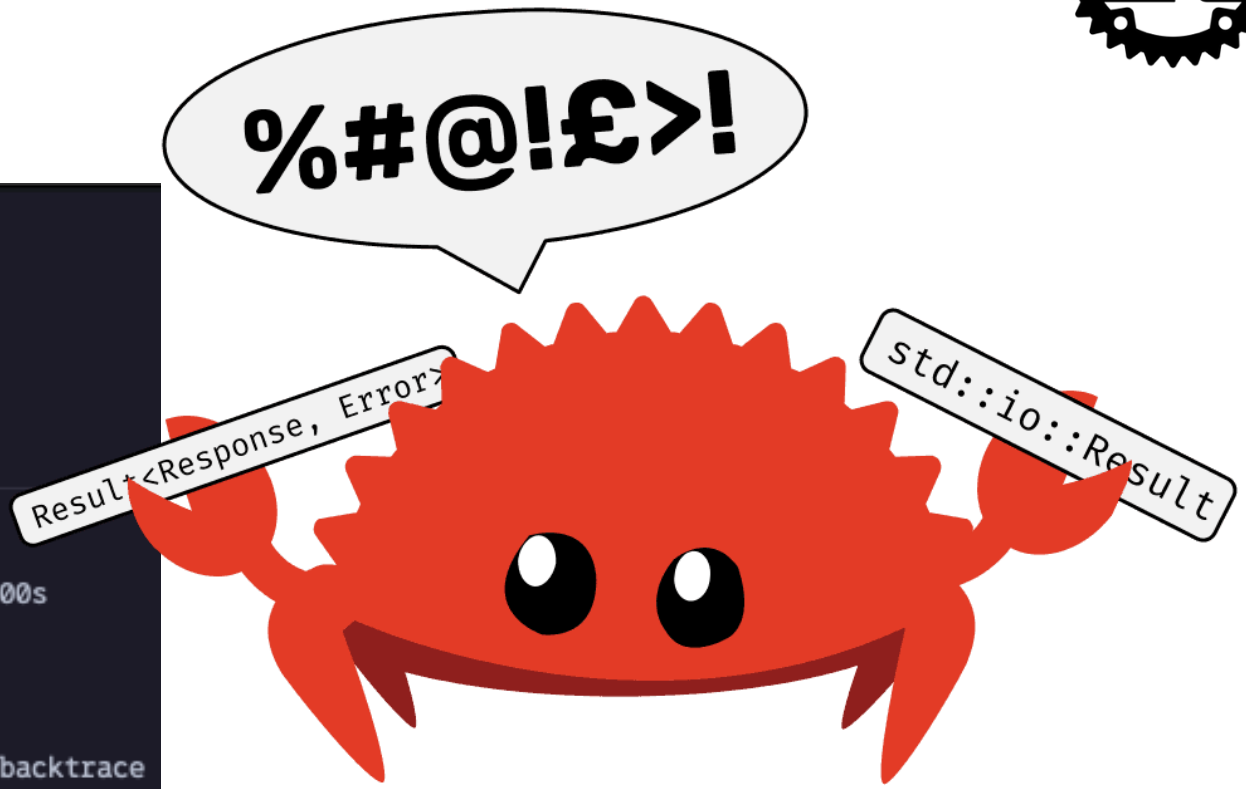


```
fn main() {
    let vector = vec![1, 2, 3, 4, 5];
    let test = vector[5];
}


```

```
warning: `panic` (bin "panic") generated 1 warning
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/panic`

thread 'main' panicked at src/main.rs:3:22:
index out of bounds: the len is 5 but the index is 5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```



Ownership – Borrowing & Move Semantik

Rust verwendet ein Ownership-System, um Speicher sicher und effizient verwalten, ganz ohne Garbage Collector.

Jeder Wert hat genau einen Besitzer, und sobald dieser aus dem Scope fällt, wird der Speicher freigegeben.

Ownership – Borrowing & Move Semantik

- Ownership wird moved:

```rust
fn print_heap_value_without_returning_ownership(string: String) {
    println!("The function now has ownership over the string, it is moved into here: {string}");
}
```

- Immutable Borrow:

```rust
fn print_heap_value_by_borrowing(string: &String) {
    println!("The function borrowed a readonly reference of the string: {string}");
}
```

- Mutable Borrow:

```rust
fn mutable_borrow(string: &mut String) {
    *string = String::from( s: "I have been mutated");
}
```

Patterns & Matching

Rusts Pattern Matching ermöglicht es, komplexe Datenstrukturen wie Enums oder Structs präzise zu analysieren und zu verarbeiten.

Vergleichbar mit match in funktionalen Sprachen.

```
match msg {
    Message::Quit => { ... }
    Message::Move { x :i32 , y :i32 } => { ... }
    Message::Write(text :String ) => { ... }
    Message::ChangeColor(r :u8 , g :u8 , b :u8 ) => { ... }
}
```

```
let Some(x :u8 ) = number else {
    println!("Kein Wert vorhanden");
    return;
};
```

Typestate Programming

Beim Typestate-Pattern wird der Zustand eines Ojekts im Typen-System codiert.

So stellt der Compiler sicher, dass Methoden nur im richtigen Zustand aufgerufen werden können.

```rust
fn main() {
    // Start disconnected
    let conn = ConnectionBuilder::new()
        .connect()  // only after this, you can build
        .build();

    // Now it's safe to use
    conn.send("Hello typestate!");
}
```

Spawns & Channels

Bei diesem Feature besteht die Möglichkeit, Nebenläufigkeit sicher und einfach umzusetzen.

Dabei kommen zwei wichtige Konzepte zum Einsatz:
spawn zum Starten von Threads
und *channel* zur Kommunikation zwischen ihnen.

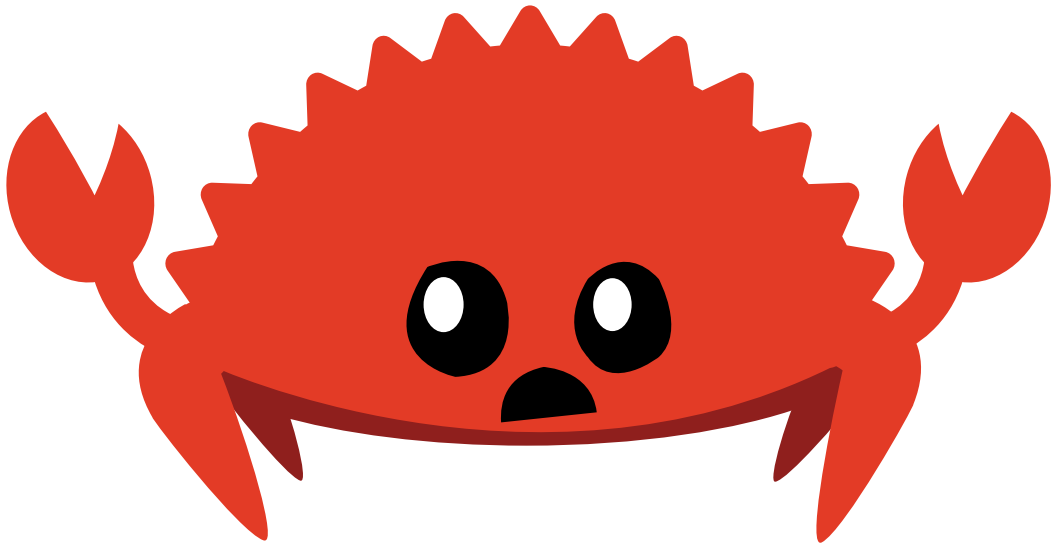
```rust
// Erstelle einen Kanal (Sender, Empfänger)
let (tx, rx) = mpsc::channel();

// Starte einen neuen Thread und sende eine Nachricht
thread::spawn(move || {
    let message = "Hallo von einem anderen Thread!";
    println!("Child-Thread: sende Nachricht…");
    tx.send(message).unwrap(); // sendet über den Kanal
    thread::sleep(Duration::from_secs(1));
    println!("Child-Thread: fertig.");
});

// Haupt-Thread wartet auf Nachricht
println!("Haupt-Thread: warte auf Nachricht…");
let received = rx.recv().unwrap();
println!("Haupt-Thread: erhalten -> {}", received);
```

Team-Fazit

- Gute Sprache?
  - Ja, aber vor Allem Trendy
- Regelmässiger Einsatz?
  - Hobby-Projekte
- Ja? Für was?
  - High-Performance Anwendungen
  - Embedded Software
  - Dev-Tooling
  - Viele Open-Source Projekte
- Nein? Wieso nicht?
  - Komplexität
  - Team Knowledge



Fazit - Kevin

- Keine Black Magic!
- Sehr explizit
- Statements vs. Expressions
- Immutable by Default
- Guter Einstieg durch Buch
- Trotz fanatischer Anhänger und "Beliebtheit" fast keine Stellen zu finden in der Schweiz

Fazit - Livio

- Einige ähnliche Konzepte wie bei C++
- Compiler erkennt viele potenzielle Fehler bereits zur Compile-zeit
- Wertvolle Erfahrung
- Kommt im beruflichen Umfeld wohl eher selten zum Einsatz