8.

Enums

Enums

```
enum Direction {
    North,
    West,
    South,
    East,
}

let d = Direction::North; // : Direction

let german = match d {
    Direction::North => "Norden",
    Direction::West => "Westen",
    Direction::South => "Süden",
    Direction::East => "Osten",
};
```

- Auflistung unterschiedlicher „Möglichkeiten“
- „Möglichkeiten“ werden ***Variants*** genannt

- Variants ***nicht*** in global Namespace
  - Via **`Typ::`** genannt
- Beispiel auch in C/Java möglich

Enums sind auch Typen!

```
impl Direction {
    fn snow_please() -> Self {
        Direction::North
    }

    fn unit_vector(&self) -> Point {
        match *self {
            Direction::North => Point { x:  0.0, y:  1.0 },
            Direction::West  => Point { x: -1.0, y:  0.0 },
            Direction::South => Point { x:  0.0, y: -1.0 },
            Direction::East  => Point { x:  1.0, y:  0.0 },
        }
    }
}

Direction::snow_please().unit_vector();
```

- Sind wie Structs auch Typen
- Können **`impl`**-Blöcke und Methoden besitzen
- Können **`#[derive]`** nutzen

Beispiel

```rust
// std::cmp::Ordering
pub enum Ordering {
    Less,
    Equal,
    Greater,
}

impl Ordering {
    pub fn reverse(self) -> Self {
        match self {
            Ordering::Less => Ordering::Greater,
            Ordering::Equal => Ordering::Equal,
            Ordering::Greater => Ordering::Less,
        }
    }
}
```

Enums

```
enum CssColor {
    /// No additional data
    None,
    /// Additional, anonymous data
    Name(String),
    /// Additional, named data
    Rgb { r: u8, g: u8, b: u8 },
}

let a = CssColor::Name("blue".into());
let b = CssColor::Rgb {
    r: 0,
    g: 255,
    b: 0,
};
```

- Variants können weitere Daten halten
- Zugriff via Destructuring
  - Pattern ist refutable!

```
match a {
    CssColor::None => {}
    CssColor::Name(ref s) => {
        println!("{}", s);
    }
    CssColor::Rgb { r, .. } => {
        println!("{}", r);
    }
}
```

Beispiel

```rust
// std::net::IpAddr
pub enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}

impl IpAddr {
    pub fn is_loopback(&self) -> bool {
        match *self {
            IpAddr::V4(ref a) => a.is_loopback(),
            IpAddr::V6(ref a) => a.is_loopback(),
        }
    }
}
```

Algebraische Datentypen

- Typen als Mengen ansehen
  - Endliche Typen: `bool`, `i32`, …
  - Unendliche Typen: `String`
- Eigene Typen bestehen aus existierenden Typen
  - Unterschiedliche Arten der Kombination
- Summentypen: Vereinigung der Mengen
  - Enums
- Produkttypen: Kartesisches Produkt der Mengen
  - Structs, Tuple, …

Algebraische Datentypen

```rust
enum Sum {
    A(Ordering),
    B(bool),
}
```

```rust
struct Product {
    a: Ordering,
    b: bool,
}
```

```rust
pub enum Ordering {
    Less,
    Equal,
    Greater,
}
```

bool	Ordering	Sum	Product
• true • false	• Less • Equal • Greater	• A(Less) • A(Equal) • A(Greater) • B(false) • B(true)	• { a: Less, b: false } • { a: Equal, b: false } • { a: Greater, b: false } • { a: Less, b: true } • { a: Equal, b: true } • { a: Greater, b: true }

Option

```rust
enum Option<T> {
    None,
    Some(T),
}

// This is not the real code for HashMap!
// Here, `Key` and `Value` are types
// used inside the HashMap.
impl HashMap {
    pub fn get(&self, key: Key)
        -> Option<Value>
    { … }
}
```

- Repräsentiert einen *möglichen* Wert
- Ist generisch über **T**
- In Rust gibt es kein null
  - Referenzen sind immer gültig
- Mögliche Abwesenheit eines Wertes immer explizit mit **`Option<T>`** gekennzeichnet!
- „The Billion Dollar Mistake“

Result

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}

// You can leave out parts of the type
// annotation.
let parse_result: Result<i32, _> =
    "hello".parse();

match parse_result {
    Ok(value) => {} // use `value`
    Err(e) =>
        println!("Invalid input! Details: {}", e),
};
```

- Repräsentiert Berechnung eines Wertes, die misslingen kann
- Ist generisch über **T** und **E**
- Wird genutzt, wenn über die Abwesenheit eines Wertes mehr Informationen existieren

Mehr im Kapitel „Error Handling“!