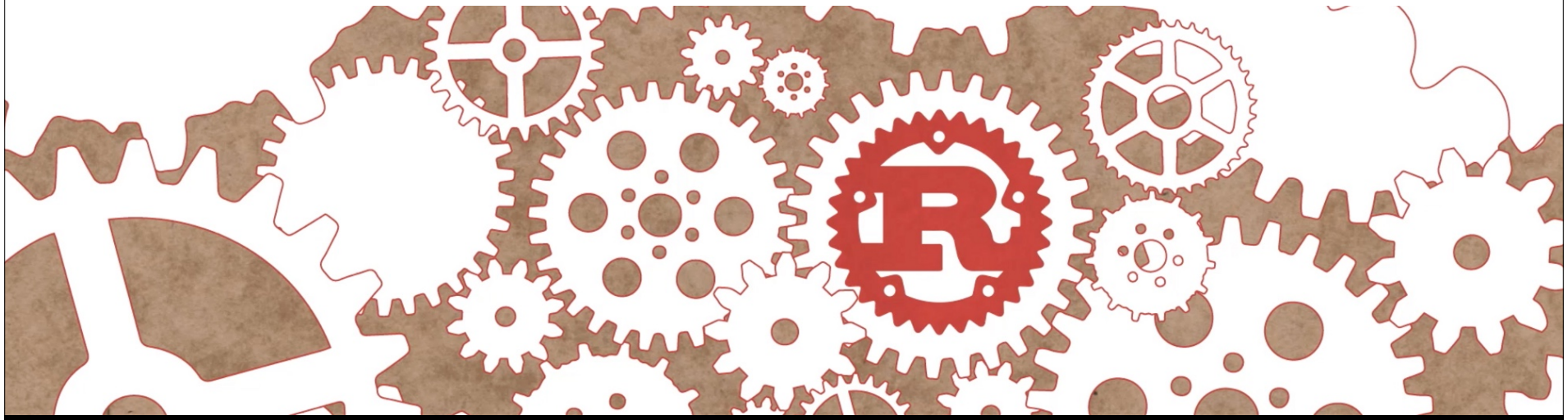
Vorlesung

Rust

Grundlagen

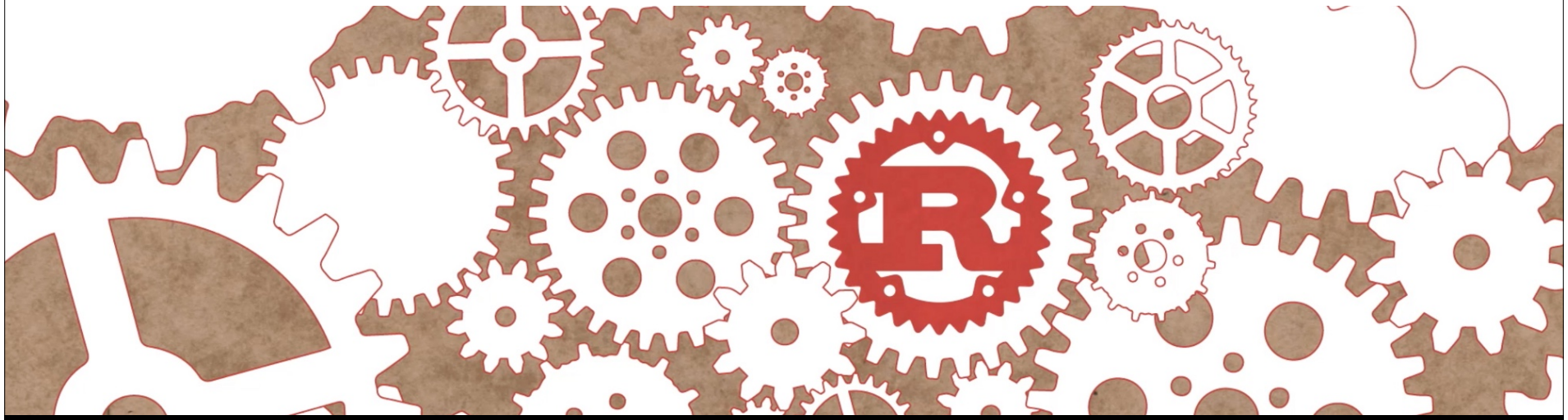
Rust Grundlagen

1. **Hello World**
2. **Primitve Typen & Ausgabe**
3. **Kontrollstrukturen und Funktionen**
4. **Expression vs. Statement**
5. **Kommentare & Codestil**



Rust Grundlagen

1. **Hello World**
2. Primitve Typen & Ausgabe
3. Kontrollstrukturen und Funktionen
4. Expression vs. Statement
5. Kommentare & Codestil



Hello World

hello.rs

```
fn main() {
    println!("Hello World");
}
```

```
$ rustc hello.rs
$ ls
hello.rs hello
$ ./hello
Hello World!
```

`fn main() {`
 	
`}`
	• **`fn`** definiert Funktion
	• **`main`** ist Einstiegspunkt

`println!("Hello World");`	• **`println!`** gibt Zeile aus • **`!`** bedeutet Macro • **`;`** beendet Anweisung

Online Compiler

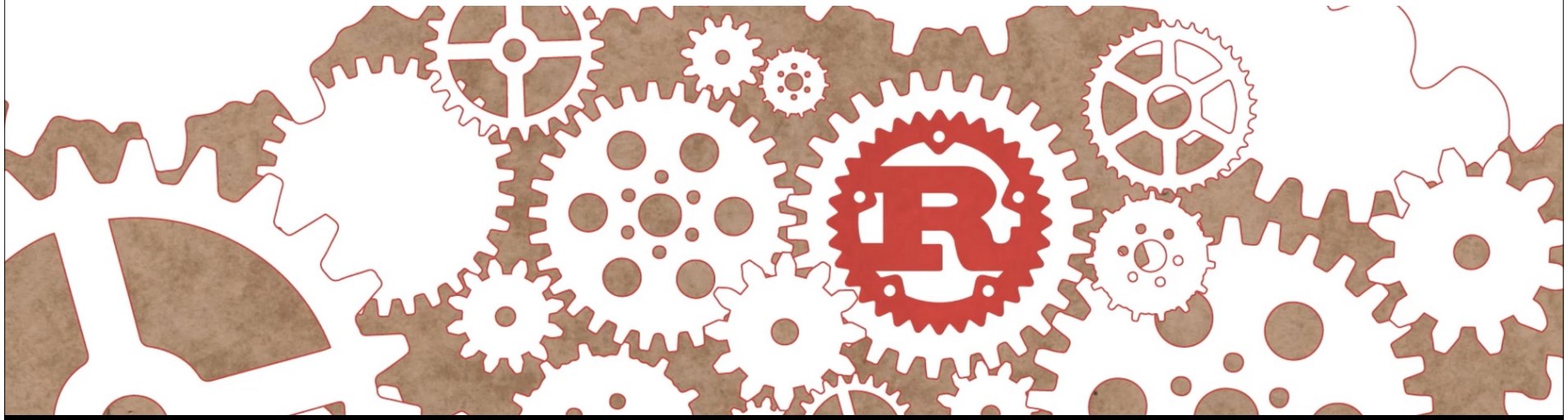
Variablen

- Mit `let` ein “Variable Binding” erstellen
  - Statisch typisiert? Wo sind die Typen? → Typinferenz!
- Variablen sind **immutable by default** (nicht veränderbar!)
  - Mit `mut` Keyword als mutable deklarieren

```
let a = 3;
let b = 3.14;
let c = true;
a = 4;  // error: re-assignment of immutable variable `a`
let mut x = 3;
x = 4; // ok
```

Rust Grundlagen

1. Hello World
2. **Primitve Typen & Ausgabe**
3. Kontrollstrukturen und Funktionen
4. Expression vs. Statement
5. Kommentare & Codestil



Primitive Typen

Integer	Fließkomma	Andere
Feste Größe, Vorzeichen: **`i8, i16, i32, i64`**	"float": **`f32`**	Boolean (true o. false): **`bool`**
Feste Größe, nur positiv: **`u8, u16, u32, u64`**	"double" **`f64`**	Unicode Skalar (32 bit): **`char`**
"*pointer sized*" (variable): **`isize, usize`**		String slice (später mehr) **`str`**

Typumwandlung / Casten mit Keyword **`as`**

```
let x = 3i32;
let y = x as u16;
```

Primitive Typkonstuktoren

Tupel

`( T, U, …)`

- Heterogene, endlichen Sequenz
- Länge/Arität fest zur Kompilierzeit
- Beispiele:

  `(u8, bool)`

  `(u64, char, i8)`

  `(T,)` ← Tupel mit einem Element

  `()` ← “void”
- Zugriff mit .0, .1, usw. (oder destructure!)

Arrays und Slices

`[ T; N ]`

- Homogene Sequenz
- Länge N fest zur Kompilierzeit!
- Beispiele:

  `[bool; 3]`

  `[u32; 8]`

  `[T; 1]` ← Array mit einem Element
- Zugriff mit **`[0]`**, **`[1]`**, usw.
- **`[T]`** → Slice: „View“ in Speicherblock, z.B. Array (später mehr)

Structs

- Syntax wie in C
  - Über Structs lassen sich auch neue Typen erstellen
  - ‘Vergleichbar’ mit Java-Klassen (später mehr)

```
struct Point {
    x: f32,
    y: f32,
}

struct Student {
    id: u32,
    grade: f32,
}
```

```
let point = Point {
    x: 5.0,
    y: 3.14,
};

let peter = Student {
    id: 123456,
    grade: 3.14,
};
println!(“ID-{}: {}”,
         peter.name,peter.grade);
```

Beispiele: Typen

```
// Explicit type annotations with `: T` (rarely necessary!)
let a:       = true;
let b:       = '水'; // Unicode :)

let c:      = 3;
let d:      = 3.1
                  // ^^^ called {integer} and {float} in error messages

let t:                = ('♥', true);
let (x, y): (char, bool) = ('♥', true); // destructuring …
let (u, v) = t; // this works, too

t.0 == x; // accessing tuple elements, both sides are the same
t.1 == y;
```

Ausgabe

- Makro-Familie: `println!() print!(), format!(), …`
  - `Rust ab 1.32: dbg!()`
- Erstes Argument: Formatstring mit Platzhaltern
- Danach: Werte für Platzhaltern

```
println!("Kein Platzhalter hier …");

let a = 3;
println!("Der Wert von a ist {}", a);

let b = true;
println!("a ist {} und b ist {}", a, b);

let arr = [3, 1, 4];
println!("arr ist {}", arr);
```

- {} → *user-facing output* (standard)
- {:?} → Debug-Ausgaben
- {:#?} → fancy Debug-Ausgaben

std::fmt Doku

```
std::fmt::Display` is not satisfied
 --> <anon>:3:20
  |
9 |     println!("{}", arr);
  |                    ^^^
[…]
  = note: `[{integer}; 3]` cannot be formatted with the
default formatter; try using `:?` instead if you are using
a format string
  = note: required by `std::fmt::Display::fmt`
```

Beispiele: Typen & Arrays

```rust
// Fixed size arrays, size in type (→ size fixed at compile time)
let a: [i32; 3] = [2, 4, 6];
let b: [char; 2] = ['☂', '🐴'];

// we can call methods on arrays, and index with []
println!("{} and {}", a.len(), b.len()); // output: „3 and 2“
println!("{} and {}", a[0], b[1]); // output: „2 and 🐴“

let c: [char; 5] = ['a', 'b', 'c', 'd', 'e'];

// Slices: size not in type, but a runtime value
let d:         = &c[1..4];
let e:         = &c[3..];
let f:         = &c[..];

println!("{}; {}; {}", d.len(), e[0], f[4]); // output: „3; d; e“
```

- Methoden auch auf primitive Typen aufrufbar

```rust
4.is_power_of_two();
// true
```

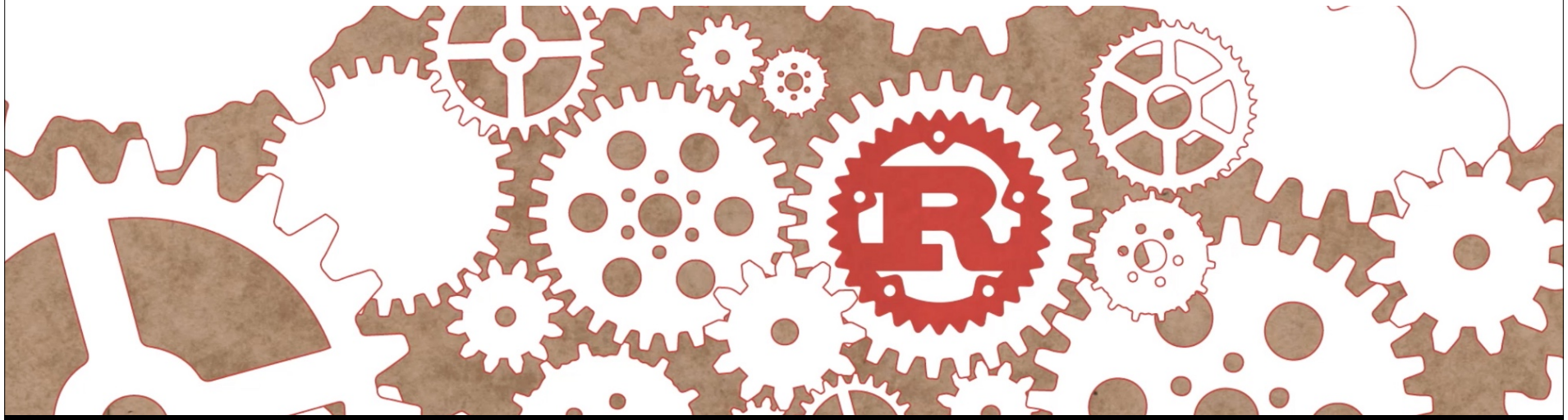
Konstanten und Statics

- Explizite Typannotation nötig
- **Konstanten**
  - Keine feste Adresse
  - In `SCREAMING_SNAKE_CASE`
- Initialisierung nur mit constant expression
- **Static**
  - Fest Adresse im Speicher
  - Sehr selten nötig

```
const PI: f64 = 3.1415926;
const START_POINT: Point = Point {
    x: 10.0,
    y: 7.0,
};

// error: function calls not allowed!
const HOME: Point = Point::origin();

// You rarely need this
static FOO: bool = true;
```

Rust Grundlagen

1. Hello World
2. Primitve Typen & Ausgabe
3. **Kontrollstrukturen und Funktionen**
4. Expression vs. Statement
5. Kommentare & Codestil



if & else & while & loop

- Bedingung ohne runde Klammern
- Rumpf zwingend mit geschweiften Klammern!
- `break`; und `continue`; funktionieren wie gewohnt in allen Schleifentypen

```
if a == 4 {
    println!("If branch");
} else if a > 10 {
    println!("Else-If branch");
} else {
    println!("Else branch");
}

while a < 10 {
    a += 1;
}

loop { // equivalent to `while true { }`
    println!("yolo!");
}
```

Funktionen

- Erst Parametername, dann –typ
- Freie Funktionen (kein Empfängerobjekt, wie z.B. `this`)
- Definition in anderen Funktionen möglich!
- Typinferenz zaubert wieder!
- Was ist mit Überladung?
  - Gibt's nicht!
    - (aus gutem Grund)

```
fn foo() { } // does nothing

fn print_number(n: i64) {
    println!("A number: {}", n);
}

fn print_sum(a: i32, b: i32) {
    println!("A sum: {}", a + b);
}

fn main() {
    foo();
    print_number(20);
    print_sum(20, 22);
}
```

Funktionen

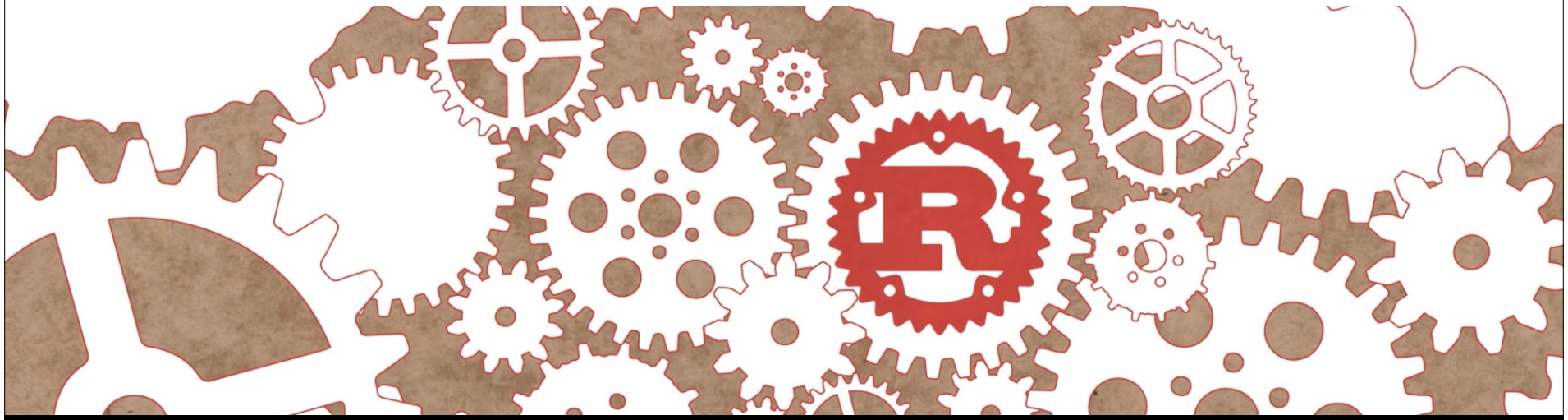
- Rückgabetyp hinter "→"
- Kein return nötig (idiomatisch!)
  - „*Everything is an expression*"
  - Aber möglich (insb. für „*early return*")
- Zwei Werte zurückgeben?
  - → Tuple

```
fn square(n: i32) → i32 { // returns i32
    n * n    // no "return" keyword?!
}

fn is_prime(n: u64) → bool {
    if x ⩽ 1 {
        return false;
    }
    // lots of code calculating
       `prime: bool`
    prime
}

fn double_triple(n: i32) → (i32, i32) {
    (2 * n, 3 * n)
}
```

```
fn main() {
    println!("3*3 = {}", square(3));
    let (double, triple) = double_triple(7);
}
```

Rust Grundlagen

1. Hello World
2. Primitve Typen & Ausgabe
3. Kontrollstrukturen und Funktionen
4. **Expression vs. Statement**
5. Kommentare & Codestil



Expression vs. Statement

- **Expression**: Gibt Wert zurück, „ergibt ausgewertet etwas“
  - Literale: `27` | `"hallo"` | `true`
  - Operationen: `27 + 3` | `a + b` | `true && false`
  - Funktionsaufrufe: `foo()` | `square(3)`
  - Alles andere… *Außer*:
- **Statements**: „Ergeben nichts“ („nichts“ ∉ {`void`, `bottom`, `null`})
  - `let` Bindings
  - Semikolon wandelt Expression in Statement:

```
any_expression; // a statement
```

- *Wofür ist das sinnvoll?*

if-else Expression

- Nur möglich, wenn else Zweig vorhanden
- Alle Zweige müssen den selben Typen zurückgeben
- Vorsicht mit den Semikola!

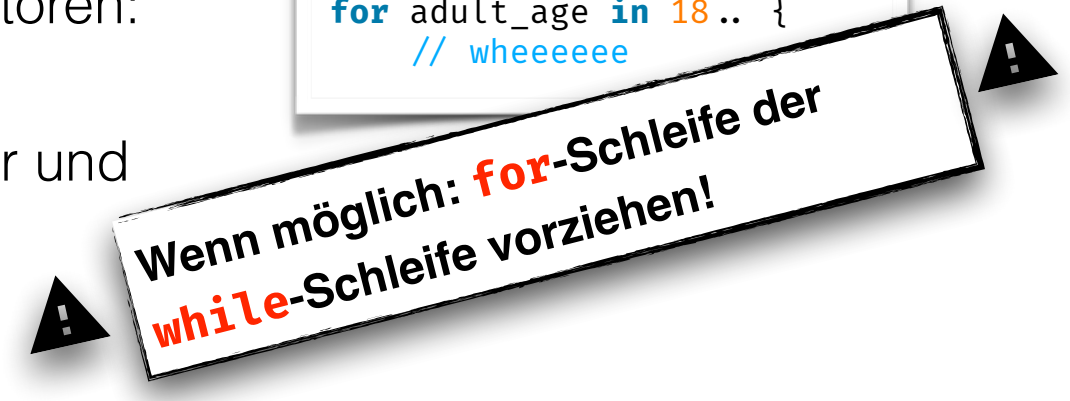
```
let a = 5;
let b = if a ⩾ 50 { 100 } else { 0 };

let c = if b % 2 == 0 {    // type of c?
    do_some_work();
    'w'
} else {
    do_some_other_work();
    's';
};

fn absolute_value(n: i32) → i32 {
    if n < 0 { -n } else { n }

note: there is already n.abs()
no need to write it yourself
```

```
error[E0308]: if and else have incompatible types
  --> <anon>:4:13
   |
4  | let c = if b % 2 == 0 { // type of c?
   |             ^ expected char, found ()
   |
   = note: expected type `char`
   = note: found type `()`
```

for-Schleife (mit Referenz)

- Syntax:
  - **`for var_name in expression { code }`**
- var_name muss ein Pattern (wie bei „let pattern =“) sein:
  - **`for &(a, b) in &[(2, 4), (3, 9)] { … }`**
  - **‘ _ ‘ → wenn var nicht benutzt wird.**
- expression muss „ein Iterator“ sein
  - Unterschiedliche Iteratoren: Ranges, Container, …
  - In Kapitel „Traits“ mehr und genaueres dazu

```rust
for i in 1..10 {
    println!("{}", i);
}

for _ in 1..10 {
    println!("Hello");
}

let arr = [3, 27, 42];
for elem in &arr {
    println!("{}", elem);
}

for adult_age in 18.. {
    // wheeeeee
```



Referenz

Kommentare und Codestil

```
// single line comments
code;

// Multiline comments are
// written like this. You shall
// not use /* */ comments ;-)
code;

/// Three slashes to start a
/// doc-comment for ‘pub fn'
///
/// Comments, doc-comments in
/// particular, are written in
/// Markdown. This is
/// important for the rendered
/// docs.
pub fn this_func_has_docu() {}

//! If you want to describe the
//! parent item  (e.g. the
//! module) instead of the
//! following, use these
//! comments.
```

Namen

- `snake_case`
  - Variablen
  - Funktionen/Methoden
  - Macros
  - Crates/Module
- `UpperCamelCase`
  - Typen
  - Traits
  - Enum Variants
  - (TcpSocket nicht TCPSocket!)
- `SCREAMING_SNAKE_CASE`
  - Konstanten
  - Statische Variablen

Diverses

- Öffnende, geschweifte Klammer nicht in eigene Zeile
- Schließende, geschweifte Klammer immer in eigene Zeile (außer `else`)
- Mit 4 Leerzeichen einrücken
- Abschließende Kommata in Listen über mehrere Zeilen
- Kein `return` wenn nicht nötig!

Beachte: Style Guide wird derzeit noch formuliert!

Disclaimer / Quellen

Dieses Material ist für das Labor Betriebssysteme, Studiengang AIN an der HTWG Konstanz erstellt worden.

Der Inhalt basiert auf den Folien von Lukas Kalbertodt:

https://github.com/LukasKalbertodt/programmieren-in-rust