

# 1.

---

## Rust Grundlagen

---

- Spracheigenschaften
- Warnung: Lernkurve
- Hello World (& Ausgabe)
- Primitive Typen
- Kontrollstrukturen
- Funktionen
- Expression vs. Statement
- Kommentare & Codestil

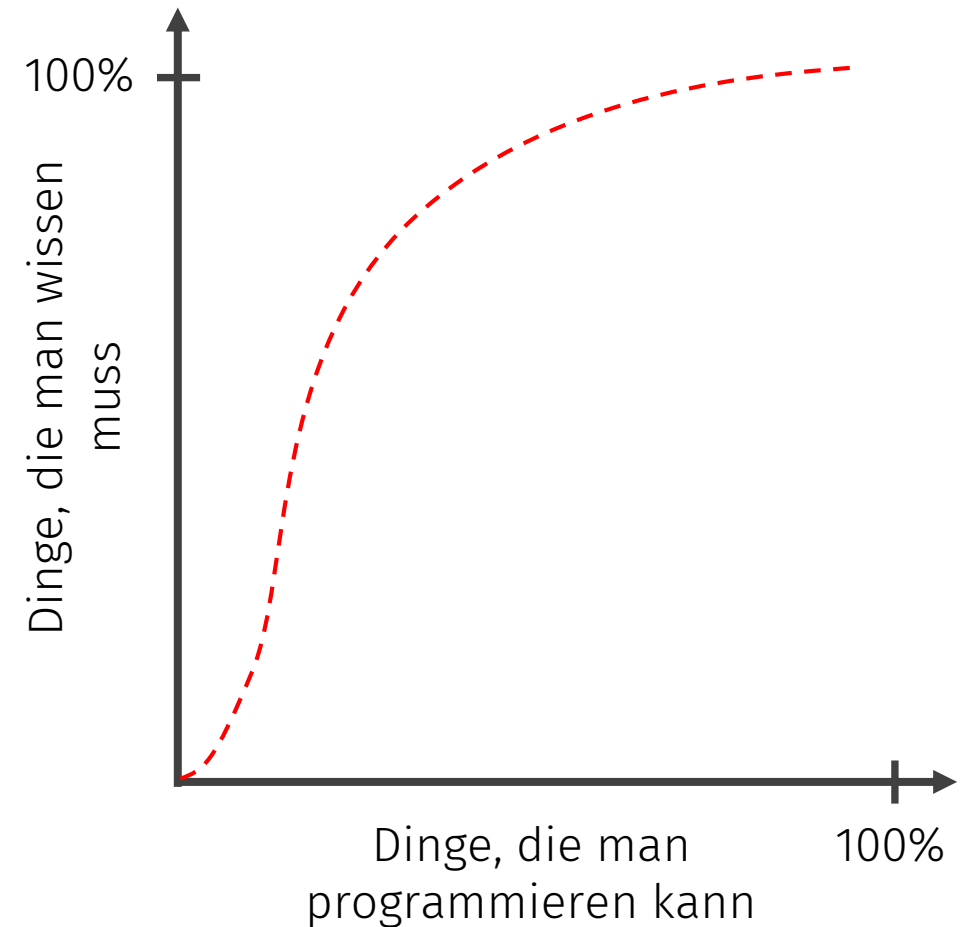
# Spracheigenschaften

- Multiparadigmensprache
- Kompiliert
- *Typisierung*: Statisch, Stark, Linear, Inferiert
- Cross-Plattform: Linux, Windows, OS X, Android, ...
- Open Source (Community Project)



# Ein Wort der Warnung: Lernkurve

- Frustrationsfaktor „steile Lernkurve“
  - [Umfragen bestätigen Vermutung](#)
  - Schwierig zu lernen und lehren
- ➔ Nicht verunsichern lassen
- ➔ „Dinge erstmal hinnehmen“



# Hello World

Online-Compiler:

<https://play.rust-lang.org>

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 (erstmal: vergleichbar mit Funktion)
- **;** „beendet Anweisungen“

# Variablen

```
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
```

- 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

# Primitive Typen

## Integer

- Feste Größe, Vorzeichen:  
`i8, i16, i32, i64`
- Feste Größe, nur positiv:  
`u8, u16, u32, u64`
- „*pointer sized*“ (variable):  
`isize, usize`

## Fließkomma

- „float“:  
`f32`
- „double“:  
`f64`

## Andere

- Boolean (`true` o. `false`):  
`bool`
- Unicode Skalar (32 bit!):  
`char`
- String slice (später mehr):  
`str`

Typumwandlung/Casten mit  
Keyword „`as`“

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

# Primitive Typkonstruktoren

## Tupel

`(T, U, ...)`

- Heterogene, endliche Sequenz
- Länge/Artität 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)

# Beispiele: Typen

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

let c: i32 = 3;      // also possible: u32, i8, usize, ...; often determined later
let d: f64 = 3.14;  // also possible: f32; often determined later
                    // ^^ called {integer} and {float} in error messages

let t: (char, bool) = ('♥', 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!()`, ...
- Erstes Argument: Formatstring mit Platzhaltern
- Danach: Werte für Platzhalter

```
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](#)

```
error[E0277]: the trait bound `[{integer}; 3]:
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

```
// 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!("{}", a.len(), b.len()); // output: „3 and 2“
```

```
println!("{}", 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: &[char] = &c[1..4]; // ['b', 'c', 'd']
```

```
let e: &[char] = &c[3..]; // ['d', 'e']
```

```
let f: &[char] = &c[..]; // ['a', 'b', 'c', 'd', 'e']
```

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

Methoden auch auf primitiven Typen aufrufbar (später mehr):

```
16.is_power_of_two();  
// true
```

# if & else & while & loop

```
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!");  
}
```

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

# Funktionen

```
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);
}
```

- 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)

# Funktionen

```
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)
}
```

- Rückgabebetyp hinter „->“
- Kein **return** nötig (idiomatisch!)
  - „*Everything is an expression*“
  - Aber möglich (insb. für „*early return*“)
- Zwei Werte zurückgeben?  
➔ Tuple

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

# 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“  $\notin$  {`void`, `bottom`, `null`})
  - `let` Bindings
  - Semikolon wandelt Expression in Statement:  
`any_expression; // a statement`
- Wofür ist das sinnvoll?

# if-else Expression

```
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 }
}

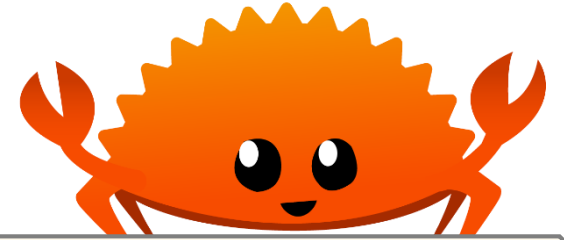
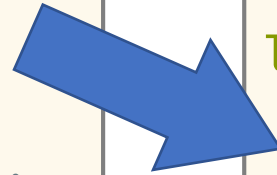
// note: there is already n.abs()
// no need to write it yourself
```

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

```
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 `()`
```

# Collatz ♥

```
let mut number = read_number();  
  
while number != 1 {  
    if number % 2 == 0 {  
        number = number / 2;  
    } else {  
        number = number * 3 + 1;  
    }  
}
```



```
let mut number = read_number();  
  
while number != 1 {  
    number = if number % 2 == 0 {  
        number / 2  
    } else {  
        number * 3 + 1  
    };  
}
```



# for-Schleife

Wenn möglich: **for**-Schleife der **while**-Schleife vorziehen!

```
for i in 1..10 {  
    println!("{}", i);  
}  
  
let arr = [3, 27, 42];  
for elem in &arr {  
    println!("{}", elem);  
}  
  
for adult_age in 18.. {  
    // wheeeeeee  
}
```

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)] { ... }
```
- `expression` muss „ein Iterator“ sein
  - Unterschiedliche Iteratoren: Ranges, Container, ...
  - In Kapitel „Traits“ mehr und genaueres dazu



# 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
///
/// Comments, doc-comments in particular,
/// are written in Markdown. This is
/// important for the rendered docs.
fn this_function_has_documentation() {}

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

[Mehr Info](#)

## Namen

- **snake\_case**
  - Variablen
  - Funktionen/Methoden
  - Macros
  - Crates/Module
- **UpperCamelCase**
  - Typen
  - Traits
  - Enum Variants
  - (TcpSocket nicht TCPSocket!)
- **SCREAMING\_SNAKE\_CASE**
  - Konstanten
  - Statische Variablen

[Mehr Info](#)

## 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:** Styleguide wird derzeit noch formuliert

**Bitte idiomatischen Code schreiben!**