

RUST

Tibor Kalman

31.01.2022,

19.06.2022,

20.06.2022,

21.06.2022 ,

22.06.2022 ,

23.06.2022,

09.07.2022 ,

12.07.2022,

13.07.2022,

15.07.2022,

16.07.2022 ,

17.07.2022,

21.07.2022,

23.07.2022,

24.07.2022,

30.07.2022,

31.07.2022,

03.08.2022,

07.08.2022,

11.08.2022

13.08.2022

17.08.2022

19.08.2022

20.08.2022

27.08.2022

28.08.2022

03.09.2022

10.9.2022

24.09.2022

1.10.2022

15.10.2022

06.11.2022

13.11.2022

20.11.2022

02.03.2024

<https://rust-lang-de.github.io/rustbook-de/ch19-06-macros.html>

<https://rust-lang-de.github.io/rustbook-de/ch20-00-final-project-a-web-server.html>

TODO: <https://stackoverflow.com/questions/69868409/how-do-i-split-my-rust-program-into-many-files>

TODO: komplexere Enums beschrieben?

TODO: Module beschrieben?
TODO: <https://doc.rust-lang.org/reference/>
TODO: Tests
TODO: RC
TODO: RefCell
TODO: Speicherlecks mit RC und RefCell
TODO: 1. Each value is owned by a variable
2. when the owner goes out of scope the value will be deallocated
3. There can only be one owner at a time
TODO: Explain RAII
TODO:
let _string_slice = &string[12..]; // give all after the 12th byte, not after the tenth character
TODO: shadowing: wenn in { } nochmal die dieselbe Variablenamen verwendet wird
TODO: usize/isize 32 bit on 32 bit architecture and 64 bit on 64bit architecture
TODO: RUST is using snake_case
TODO: cargo expand (shows e.g. how println! Looks after compilation)
TODO: to convert a string to a number .parse
TODO: please avoid using global variables, but if you really need the use keyword „const“
TODO: elements of an enum can have diggerent types like u64, String, etc...
TODO: recoverable and unrecoverable errors
TODO: explain unimplemented!()
TODO: eprintln! beschreiben
TODO: format! ?
TODO: array slice [2..4], erster Wert inklusive, 2ter exklusive
TODO allow dead code
TODO String über for c in hello.chars() iterieren
TODO: Maybe implement here or in another document the array_of_characters examples: mini game.

dieses Dokument dient als Kursunterlage für den Kurs RUST von Tibor Kalman.
Viele Beispiele sind sicherlich aus anderen Beispielen die im Netz ebenfalls zu finden sind, angelehnt. Jedoch ist jedes Beispiel durchdacht und von mir selbst geschrieben und ausprobiert worden. Teilweise gebe ich das Entstehungsdatum mit an, falls das ein oder andere Programm in Zukunft nicht mehr funktionieren sollte (was durch Änderungen der Sprache RUST durchaus passieren kann).

Einführung:

Installation (2.3.2024):

Windows:

<https://www.rust-lang.org/learn/get-started>

```
C:\Users\MyUserName>rustc -V
```

```
rustc 1.76.0 (07dca489a 2024-02-04)
```

```
C:\Users\MyUserName>cargo --version
```

```
cargo 1.76.0 (c84b36747 2024-01-18)
```

Softwareentwickler die sich gerne mit Rust auseinandersetzen nennt man Rustaceans.
(Quelle: https://tourofrust.com/chapter_1_de.html)

Die kleine Krabbe, das Symbol von Rust nennt sich Ferris:
(Quelle: https://tourofrust.com/chapter_1_de.html)

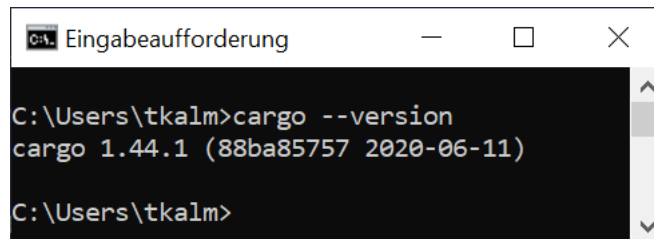
there is support for RUST in IntelliJ

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

https://www.tutorialspoint.com/rust/rust_string.htm

Rust installing 64 bit version:
A console opens under windows. Follow the instructions.

To test if it was installed correctly:



```
C:\Users\tkalm>cargo --version
cargo 1.44.1 (88ba85757 2020-06-11)
C:\Users\tkalm>
```

build project :
cargo build
cargo build --release

run project :
cargo run

Soll schneller sein als „cargo run“:
cargo check

test project :
cargo test

build documentation :
cargo doc

publish library to crates.io :
cargo publish

Zeigen:

```
cargo new projektname
```

```
C:\Users\MyUserName>rustc -V
```

```
rustc 1.57.0 (f1edd0429 2021-11-29)
```

Installation Linux:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Rust playground:

Dies ist ein wichtiger Link für Anfänger und diejenigen, die mal eben schnell etwas ausprobieren wollen, hier lassen sich Codebeispiele online ausführen.

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

TODO:

<https://www.w3adda.com/rust-tutorial>

Short Rust crash course:

<https://www.w3adda.com/rust-tutorial>

explain pub

Befehl zum formatieren des Codes in der Shell

```
rustfmt
```

Befehl zum Bauen, z.B.:

```
rustc main.rs
```

Updaten Rust (hat am 2.3.2024 nicht mehr funktioniert):

```
rustc --version
```

rustup update

Rust deinstallieren:

```
rustup self uninstall
```

Cargo update (hat am 2.3.2024 nicht mehr funktioniert):

```
cargo update
```

```
cargo install cargo-edit
```

IntelliJ

Rust lässt sich auch prima mit IntelliJ entwickeln.

TODO: add of plugin.

1 Beispiel Hello World:

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

Das Semikolon kann man auch weglassen:

```
fn main() {  
    println!("Hello, world!")  
}
```

Ausgabe:

Hello, world!

Bemerkung:

Das Ausrufezeichen nach dem println bedeutete lediglich das println ein Makro ist und keine Funktion. TODO.

2 Beispiel: mit String :

```
fn main() {  
    println!("Hallo, {}!" , "Du");  
}
```

Ausgabe:

```
Hallo, Du!
```

3 Beispiel String mit einigen Operationen:

```
fn main() {  
    let mut mein_string = String::from("Mein Text");  
    println!("mein_string a:{}", mein_string);  
  
    mein_string = String::from("Mein Text");  
    println!("mein_string a1:{}", mein_string.to_uppercase());  
  
    mein_string = String::from("Mein Text");  
    println!("mein_string a2:{}", mein_string.to_lowercase());  
  
    mein_string = mein_string + " wird länger";  
    println!("mein_string b: {}", mein_string);  
  
    mein_string.push_str(" und noch etwas länger");  
    println!("mein_string c: {}", mein_string);  
  
    let mut mein_string = "    mit viel Leerzeichen davor und danach    ";  
    println!("mein_string d:{}", mein_string);  
  
    println!("mein_string trim :{}", mein_string.trim().to_owned() + String::from("->jetzt nicht mehr").as_str());  
}
```

Ausgabe:

```
mein_string a:Mein Text  
mein_string a1:MEIN TEXT  
mein_string a2:mein text  
mein_string b: Mein Text wird länger  
mein_string c: Mein Text wird länger und noch etwas länger  
mein_string d:    mit viel Leerzeichen davor und danach  
mein_string d2:mit viel Leerzeichen davor und danach->jetzt nicht mehr
```


4 Beispiel mit Ganzzahl:

```
fn main() {  
    println!("Hallo, {}!", 77);  
}
```

Ausgabe:

Hallo, 77!

5 Beispiel mit mehreren Variablen (Konstanten) in der Ausgabe:

```
fn main() {  
    println!("erster Wert: {} zweiter Wert: {} texttexttext" , 77 , "MeinZweiterWert");  
}
```

Ausgabe:

erster Wert: 77 zweiter Wert: MeinZweiterWert texttexttext

6 Beispiel mit Variablennamen: (2022-01-31)

```
fn main() {  
    println!("erster Wert: {irgendeinText} zweiter Wert: {meineZahl}", meineZahl=77, irgendeinText="MeinZweiterWert");  
}
```

Ausgabe:

erster Wert: MeinZweiterWert zweiter Wert: 77

Aufgabe:

Probieren Sie aus, was passiert, wenn Sie mehr geschweifte Klammern Paare einfügen, als Werte die Sie übergeben:

```
fn main() {  
    println!("erster Wert: {} zweiter Wert: {} texttexttext {} " , 77 , "MeinZweiterWert");  
}
```

Lösung:

RUST kompiliert erst gar nicht.

7 Beispiel:

```
fn main() {  
    println!("erster Wert: {1} zweiter Wert: {0} texttexttext {1} ", 77, "MeinZweiterWert");  
}
```

Ausgabe:

erster Wert: MeinZweiterWert zweiter Wert: 77 texttexttext MeinZweiterWert

8 Beispiel das NICHT kompiliert:

```
fn main() {
    println!("erster Wert: {1} zweiter Wert: {0} texttexttext {1} ", 77, "MeinZweiterWert", 88);
}
```

Ausgabe:

```
C:\Users\tkalm\cargo\bin\cargo.exe run --color=always --package RustTutorialMy --bin RustTutorialMy
  Compiling RustTutorialMy v0.1.0 (C:\Users\tkalm\IdeaProjects\RustTutorialMy)
error: argument never used
--> src\main.rs:2:95
|
2 |     println!("erster Wert: {1} zweiter Wert: {0} texttexttext {1} ", 77, "MeinZweiterWert", 88);
|     ----- ^ argument never used
|
|     formatting specifier missing
error: aborting due to previous error

error: could not compile `RustTutorialMy`.

To learn more, run the command again with --verbose.

Process finished with exit code 101
```

9 Beispiel Konstante i32 ohne Angabe des Typs:

```
fn main() {  
    let m_v = 99;  
    println!("m_v: {} ", m_v);  
}
```

Ausgabe:

m_v: 99

Variablen muss man mit let initialisieren, wobei die Angabe des Variablentyps möglich ist, aber weggelassen werden sollte, da der Compiler selbst entscheidet welchen Typ er verwendet. Es wäre also auch folgende Schreibweise möglich gewesen:

```
fn main() {  
    let m_v:i8 = 99;  
    println!("m_v: {} ", m_v);  
}
```

Eine 129 hätten Sie beispielsweise m_v nicht zuweisen können, da i8 maximal bis 127 und minimal bis -128 geht. Wenn nur positive Werte verwendet werden sollen kann man auch u8 bzw uX verwenden, dann hat man Zahlenwerte von 0..=255.

.

10 Beispiel Konstante i32 mit Angabe des Typs:

```
fn main() {  
    let m:i32 = 99;  
    println!("m: {} ", m);  
}
```

Ausgabe:

m: 99

Nachdem der Variablen m ein Wert zugewiesen wurde, lässt er sich nicht wieder ändern. Es handelt sich nach der Zuweisung also um eine Konstante.

Folgendes lässt sich **nicht** kompilieren:

```
fn main() {  
    let m:i32 = 99;  
    println!("m: {} ", m);  
  
    m = 100;  
}
```

11 Beispiel Konstante : Zuweisung wird später durchgeführt (Definition und Initialisierung nacheinander)

```
fn main() {  
    let m;  
    m = 99;  
    println!("m: {} ", m);  
}
```

Ausgabe:

m: 99

12 Beispiele einiger Typen mit Ausgabe :

```
fn main() {  
  
    //let bool_value_default:bool; // auch ein boolean muss initialisiert werden  
    //println!("{}", bool_value_default);  
  
    let bool_value_mit_typangabe :bool = false;  
    println!("bool_value_mit_typangabe {}", bool_value_mit_typangabe);  
  
    let bool_value = false;  
    println!("bool_value {}", bool_value);  
  
    let integer_32 = 35000; // Integer ohne Typangabe ist i32  
    println!("integer_32 {}", integer_32);  
  
    let integer_8: i8 = 127; // -128 bis 127  
    println!("integer_8 {}", integer_8);  
  
    let unsigned_integer_8_mit_typ_angabe_hinten = 255u8; // 0 bis 255  
    println!("unsigned_integer_8_mit_typ_angabe_hinten {}", unsigned_integer_8_mit_typ_angabe_hinten);  
  
    let floating_point: f32 = 2.1; // f8 gibt es z.b. nicht  
    println!("floating_point {}", floating_point);  
  
    let floating_point_number_f64 = 111.333335551234567890; // ohne Typangabe ist Default ist f64  
    println!("floating_point_number_f64 {}", floating_point_number_f64);  
  
    let some_string = "Main fAlschGeschribena text";  
    println!("some_string {}", some_string);  
  
    let some_string_mit_typangabe : &str = "Main fAlschGeschribena text 2";  
    println!("some_string_mit_typangabe {}", some_string_mit_typangabe);  
  
    let ein_tupel : (bool, &str, f64) = (true, "mein text", 3.14159f64); // kann aus mehreren unterschiedlichen Typen bestehen  
    println!("ein_tupel.0 {}", ein_tupel.0);  
    println!("ein_tupel.1 {}", ein_tupel.1);  
    println!("ein_tupel.2 {}", ein_tupel.2);  
  
    let m_array = [4f32, 5.5f32, 12.0f32];  
    println!("m_array[0]: {} ", m_array[0]);  
    println!("m_array[1]: {} ", m_array[1]);  
    println!("m_array[2]: {} ", m_array[2]);  
}
```

Ausgabe:

```
bool_value_mit_typangabe false  
bool_value false  
integer_32 35000  
integer_8 127  
unsigned_integer_8_mit_typ_angabe_hinten 255
```

```
floating_point 2.1
floating_point_number_f64 111.33333555123457
some_string Main fAlschGeschribena text
some_string_mit_typangabe Main fAlschGeschribena text 2
ein_tupel.0 true
ein_tupel.1 mein text
ein_tupel.2 3.14159
m_array[0]: 4
m_array[1]: 5.5
m_array[2]: 12
```

13 Beispiel Variable : Eine veränderbare Variable wird durch mut angegeben (mutable = veränderbar)

```
fn main() {  
    let mut meine_variable = "Text des Strings";  
    println!("meine_variable: {} ", meine_variable);  
}
```

Ausgabe:

meine_variable: Text des Strings

Lässt man „mut“ weg, dann hat man eine Konstante.

14 Beispiel Variable : (mutable oder immutable)

```
fn main() {  
    let z1 = 3;  
    let mut z2 = 8;  
    println!(" z1={}, z2={}", z1, z2);  
  
    z2 = 99;  
    println!(" z1={}, z2 nach Änderung={}", z1, z2);  
}
```

Ausgabe:

```
z1=3, z2=8  
z1=3, z2 nach Änderung=99
```

Lässt man „mut“ weg, dann hat man eine Konstante und die Anwendung lässt sich so nicht mehr kompilieren.

15 Beispiel String Variable (mutable):

```
fn main() {  
    let mut mein_string = String::from("irgendein String");  
  
    string_bearbeiten(&mut mein_string);  
  
    println!("mein_string :{}", mein_string);  
}  
  
fn string_bearbeiten(string_param: &mut String) {  
    string_param.push_str(" ");  
    string_param.push_str("wird bearbeitet");  
}
```

Ausgabe:

```
mein_string :irgendein String wird bearbeitet
```

Probieren Sie aus, was passiert, wenn man an den einzelnen Stellen jeweils „mut“ weglässt. Falls Sie IntelliJ verwenden, wird es Ihnen vor dem Kompilieren schon sagen, dass „mut“ fehlt.

16 Beispiel String Variable (mutable):

In folgendem Beispiel schauen wir uns an, wie Code erstmal kompiliert und erst später durch die Verwendung, also in unserem Beispiel das Hinzufügen von einem zweiten println einen Fehler angibt. Kompilieren Sie erstmal das Programm und untersuchen Sie es.

```
fn main() {  
    let mut mein_string = String::from("Hallo");  
  
    let mein_string2 = &mut mein_string;  
    let mein_string3 = &mut mein_string;  
  
    println!("mein_string3 {}", mein_string3);  
    //println!("mein_string2 {}", mein_string2);  
}
```

Ausgabe:

```
mein_string3 Hallo
```

Wenn Sie allerdings das letzte println auskommentieren erhalten Sie erst dann die Fehlermeldung („cannot borrow `mein_string` as mutable more than once at a time“).

17 Beispiel String Variable (wie vorher nur ohne mut):

```
fn main() {  
    let mut mein_string = String::from("Hallo");  
  
    let mein_string2 = & mein_string;  
    let mein_string3 = & mein_string;  
  
    println!("mein_string3 {}", mein_string3);  
    println!("mein_string2 {}", mein_string2);  
}
```

Ausgabe:

```
mein_string3 Hallo  
mein_string2 Hallo
```

Da hier kein mut verwendet wird kann das Programm ausgeführt werden.

18 Beispiel static :

Mit static lassen sich Variablen angeben, deren Lifetime die komplette Zeit überdauert, die ein Programm aktiv ist.

```
fn main() {  
    let my_static_text: &'static str = "Mein statischer Text";  
    println!("my_static_text: {}", my_static_text);  
}
```

Ausgabe:

```
my_static_text: Mein statischer Text
```

Man spricht von einem String-Literal, wenn ein String mit „&'static str“ angelegt wird. Ein solcher String steht zur Gesamtzeit eines laufenden Programms zur Verfügung.

19 Beispiel static :

```
static KOEPERTEMPERATUR: f32 = 36.7;

fn main() {
    println!("KOEPERTEMPERATUR: {}", KOEPERTEMPERATUR);
}
```

Ausgabe:

```
KOEPERTEMPERATUR: 36.7
```

Diese statische Variable lässt sich nicht ändern.

20 Beispiel static :

```
fn main() {  
    static mut KOEPERTEMPERATUR: f64 = 36.7;  
    unsafe {  
        KOEPERTEMPERATUR = 44.4;  
        println!("KOEPERTEMPERATUR {}", KOEPERTEMPERATUR);  
    }  
    // println!("KOEPERTEMPERATUR: {}", KOEPERTEMPERATUR); // kommentieren Sie dies mal aus  
}
```

Ausgabe:

```
KOEPERTEMPERATUR 44.4
```

Um statische Variablen ändern zu können muss ein unsafe-Block drumherum gebaut werden. Merken Sie sich dennoch, dies sollte man nicht tun, weil angeblich keine Speichergarantie vergeben werden kann (https://tourofrust.com/55_de.html).

21 Beispiel String-Literal mit r# # :

```
fn main() {  
    let html: &'static str =  
        r#"<html>  
            Um Sonderzeichen wie <, >, statt &lt; etc... darzustellen  
        </html>"#;  
    println!(" HTML: {}", html);  
}
```

Ausgabe:

```
HTML: <html>  
    Um Sonderzeichen wie <, >, statt &lt; etc... darzustellen  
</html>
```

So lässt sich z.B. reltic einfach Html in einem String unterbringen, ohne dass man mit < oder > arbeiten muss.

22 Beispiel Funktion :

```
fn main() {  
    println!("vor Funktionsaufruf....");  
  
    meine_funktion();  
  
    println!("...nach Funktionsaufruf");  
}  
  
fn meine_funktion(){  
    println!("..in der Funktion...");  
}
```

Ausgabe:

```
vor Funktionsaufruf....  
..in der Funktion...  
...nach Funktionsaufruf
```

Funktionen können über andere Funktionen aufgerufen werden. Wird eine selbst geschriebene Funktion nicht irgendwo aufgerufen, dann wird sie auch nicht ausgeführt.

Variablen, die z.B. in der Funktion „main“ angelegt werden, stehen nicht ohne weiteres in anderen Funktionen zur Verfügung. Ich muss diese übergeben oder als global definieren, dazu jedoch später.

23 Beispiel Funktion mit Übergabeparameter :

```
fn main() {  
    println!("vor Funktionsaufruf....");  
  
    let h: i32 = 23;  
  
    meine_funktion(h);  
  
    println!("...nach Funktionsaufruf");  
}  
  
fn meine_funktion(z:i32){  
    println!("..in der Funktion...{}", z);  
}
```

Ausgabe:

```
vor Funktionsaufruf....  
..in der Funktion...23  
...nach Funktionsaufruf
```

Wir konnten der Funktion nun die Variable (Konstante) mittels Übergabeparameter bekanntgeben, sodass mit dieser Variablen innerhalb dieser Funktion etwas gemacht werden kann.

Beachten Sie, dass die Variable innerhalb der main-Funktion h heisst, innerhalb der eigenen Funktion jedoch z genannt wurde. Dies muss man nicht so machen.

24 Beispiel Funktion mit mehreren Übergabeparametern:

```
fn main() {  
    println!("vor Funktionsaufruf....");  
  
    let z: i32 = 23;  
    let mut yps: i32 = 11;  
  
    meine_funktion(z, yps);  
  
    println!("...nach Funktionsaufruf");  
}  
  
fn meine_funktion(z: i32, ypsilon: i32) {  
    println!("..in der Funktion...{}", z);  
    println!("..ypsilon in Funktion...{}", ypsilon);  
}
```

Ausgabe:

```
vor Funktionsaufruf....  
..in der Funktion...23  
..ypsilon in Funktion...11  
...nach Funktionsaufruf
```

Eine Funktion kann mehrerer Übergabeparameter haben. Diese können unterschiedlich Typen haben.

25 Beispiel Funktion mit Übergabeparameter :

```
fn main() {  
    println!("vor Funktionsaufruf....");  
  
    let z: i32 = 23;  
    let mut yps: i32 = 11;  
  
    meine_funktion(z, yps);  
  
    println!("...yps...{}", yps);  
    println!("...nach Funktionsaufruf");  
}  
  
fn meine_funktion(z: i32, mut ypsilon: i32) {  
    ypsilon = ypsilon + 2;  
    println!("..in der Funktion...{}", z);  
    println!("..ypsilon in Funktion...{}", ypsilon);  
}
```

Ausgabe:

```
vor Funktionsaufruf...  
..in der Funktion...23  
..ypsilon in Funktion...13  
...yps...11  
...nach Funktionsaufruf
```

Beachten Sie, dass wenn Sie den Übergabeparameter in er eigenen Funktion änderbar machen möchten, können Sie ihn mit mut versehen. Er ist dann aber nur innerhalb der Funktion änderbar. (Auch hierzu später mehr)

26 Beispiel Rückgabewert :

```
fn main() {  
    println!("vor Funktionsaufruf....");  
  
    let z: i32 = 23;  
    let mut yps: i32 = 11;  
  
    let rueckgabe:i32;  
  
    rueckgabe = meine_funktion(z,yps);  
  
    println!("...yps...{}", yps);  
    println!("...nach Funktionsaufruf");  
    println!("...rueckgabe: {} ", rueckgabe);  
}  
  
fn meine_funktion(z:i32, mut ypsilon:i32) -> i32 {  
    let ypsilon_neu = ypsilon+2;  
    println!("..in der Funktion...{}", z);  
    println!("..ypsilon_neu in Funktion...{}", ypsilon_neu);  
  
    ypsilon_neu  
}
```

Ausgabe:

```
vor Funktionsaufruf....  
..in der Funktion...23  
..ypsilon_neu in Funktion...13  
...yps...11  
...nach Funktionsaufruf  
...rueckgabe: 13
```

Beachten Sie, dass der Rückgabewert ypsilon_neu ohne Semikolon am Ende der Funktion steht. Achten Sie auch auf den Pfeil (->). Sie dürfen allerdings auch return dazu schreiben.

27 Beispiel Rückgabewert mit return:

```
fn main() {  
    println!("vor Funktionsaufruf....");  
  
    let z: i32 = 23;  
    let mut yps: i32 = 11;  
  
    yps = meine_funktion(z, yps);  
  
    println!("...yps...{}", yps);  
    println!("...nach Funktionsaufruf");  
}  
  
fn meine_funktion(z: i32, mut ypsilon: i32) -> i32 {  
    //Sie duerfen hiervoor auch "return a*h+2" schreiben  
    ypsilon+2*z  
}
```

Ausgabe:

```
vor Funktionsaufruf....  
...yps...57  
...nach Funktionsaufruf
```

Wie Sie sehen wird in diesem Beispiel eine mutable Variable „yps“ übergeben und gleichzeitig als Rückgabewert wieder „yps“ zugewiesen.

28 Beispiel Funktion die ein String erzeugt:

```
fn main() {
    let zurueckgebener_string = gib_string_zurueck();
    println!("zurueckgebener_string {}", zurueckgebener_string);
}

fn gib_string_zurueck() -> &String {
    let return_wert = String::from("irgendein String");
    &return_wert
}
*/
fn main() {
    let zurueckgebener_string = gib_string_zurueck();
    println!("zurueckgebener_string {}", zurueckgebener_string);
}

fn gib_string_zurueck() -> String {
    let return_wert = String::from("irgendein String");
    return_wert
}
```

Ausgabe:

```
zurueckgebener_string irgendein String
```

Vergleichen Sie den obigen auskommentierten Code mit dem darunter stehenden Code. Der obere Code kann nicht ausgeführt werden, da die Funktion ein Lifetime erwartet (hierzu später). Probieren Sie es aus und beachten Sie was Rust beim Kompilieren dazu sagt.

29 Beispiel Einsparen der Übergabeparameter durch Kurznotation:

Beim Anlegen eines Objekts kann man sich das angeben der Übergabeparameter einsparen

```
struct Konto {
    bezeichnung: String,
    eigentuemer: String,
    guthaben: i128
}

fn main(){
    let konto:Konto = erzeuge_konto(String::from("Sparkonto"),
        String::from("Du"));

    println!("konto Eigentümer: {}", konto.eigentuemer);
}

fn erzeuge_konto(bezeichnung: String, eigentuemer: String) -> Konto {
    Konto {
        bezeichnung,
        eigentuemer,
        guthaben: 1
    }
    /*
    Konto {
        bezeichnung: bezeichnung,
        eigentuemer: eigentuemer,
        guthaben: 1
    }
    */
}
```

Ausgabe:

```
konto Eigentümer: Du
```

Wenn Sie den auskommentierten Code verwenden, anstelle des vorhandene, werden Sie sehen, dass man sich nur das schreiben der 2 Übergabeparameter sparen konnte, da beide gleich heissen. Ob das wichtig, sinnvoll, zeitsparender ist meiner Ansicht nach fraglich.

30 Beispiel Referenz :

```
use std::ops::Add;

struct Person {
    name: String,
}

fn return_person_name(person: &Person) -> &String {
    return &person.name;
}

fn main() {
    let mut person = Person { name: String::from("Ein Name"), };

    let name_referenz = &mut person.name;
    *name_referenz = String::from("Neuer Name");

    let returned_name = return_person_name(&person);

    println!("returned_name {} ", returned_name);
}
```

Ausgabe:

```
returned_name Neuer Name
```

Die Funktion gibt einen String der Referenz zurück. Beachten Sie, dass sie allerdings so innerhalb der Funktion den Namen nicht ändern können.

31 Beispiel Adresse einer Referenz :

```
fn main() {  
    let mut my_var: i64 = 33;  
    let address_of_my_var = &my_var as *const i64 as usize;  
  
    println!("Wert der Variablen {}", my_var);  
    println!("Adresse von my_var {}", address_of_my_var);  
  
    my_var = 22;  
    let address_of_my_var = &my_var as *const i64 as usize;  
  
    println!("_ Wert der Variablen {}", my_var);  
    println!("_ Adresse von my_var {}", address_of_my_var);  
}
```

Ausgabe:

```
Wert der Variablen 33  
Adresse von my_var 483690412816  
_ Wert der Variablen 22  
_ Adresse von my_var 483690412816
```

Hier haben wir gesehen, dass eine Variable einen Wert hat der sich ändern kann. Die Adresse bleibt jedoch gleich.

32 Beispiel des Operators „.“ und „&&&“ :

```
struct Konto {  
    guthaben: i64,  
    eigentuemer: String  
}  
  
fn main() {  
    let konto = Konto { guthaben: 1111, eigentuemer: "Ich".to_string() };  
    let triple_ref_konto = &&&konto;  
  
    println!("triple_ref_konto guthaben: {}", triple_ref_konto.guthaben);  
    println!("triple_ref_konto eigentuemer: {}", triple_ref_konto.eigentuemer);  
  
    println!("triple_ref_konto eigentuemer: {}", (**triple_ref_konto).eigentuemer);  
}
```

Ausgabe:

```
triple_ref_konto guthaben: 1111  
triple_ref_konto eigentuemer: Ich  
triple_ref_konto eigentuemer: Ich
```

Mit dem Punkt-Operator ist es möglich auf Variablen eines dereferenzierten Objektes zuzugreifen, ohne `***` davor schreiben zu müssen.

33 Beispiel des Operators * :

```
fn main() {  
    let my_var: i32 = 11;  
    println!("my_var: {}", my_var);  
  
    let triple_ref_my_var: &&&i32 = &&&my_var;  
    println!("triple_ref_my_var: {}", triple_ref_my_var);  
  
    let reference_of_triple_ref_my_var: &i32 = **triple_ref_my_var;  
    println!("reference_of_triple_ref_my_var: {}", reference_of_triple_ref_my_var);  
    let dereffed: i32 = *reference_of_triple_ref_my_var;  
    println!("dereffed: {}", dereffed);  
  
    ////  
  
    let double_ref_my_var: &&i32 = &&my_var;  
    println!("double_ref_my_var: {}", double_ref_my_var);  
    let reference_of_double_ref_my_var: &i32 = *double_ref_my_var;  
    println!("reference_of_double_ref_my_var: {}", reference_of_double_ref_my_var);  
  
    let simple_ref_my_var: &i32 = &my_var;  
    println!("simple_ref_my_var: {}", simple_ref_my_var);  
    let reference_of_simple_ref_my_var: &i32 = simple_ref_my_var;  
    println!("reference_of_simple_ref_my_var: {}", reference_of_simple_ref_my_var);  
}
```

Ausgabe:

```
my_var: 11  
triple_ref_my_var: 11  
reference_of_triple_ref_my_var: 11  
dereffed: 11  
double_ref_my_var: 11  
reference_of_double_ref_my_var: 11  
simple_ref_my_var: 11  
reference_of_simple_ref_my_var: 11
```

Dieses Beispiel ist nur sehr schwer zu verstehen. Es zeigt wie man Referenzen wieder dereferenziert.

34 Beispiel Lifetime:

```
struct Konto<'a> {  
    name: &'a str,  
}  
  
fn main() {  
    let mein_name = String::from("ein String");  
    let konto = Konto {  
        name: &mein_name,  
    };  
    println!("name: {}", konto.name);  
}
```

Ausgabe:

```
name: ein String
```

TODO: Erklärung Lifetime

35 Beispiel Referenz in Funktion ändern (Lifetime):

```
use std::ops::Add;

struct Person {
    name: String,
}

fn return_person_name<'a>(person: &'a mut Person) -> &'a String {
    person.name = String::from (&person.name).add(" d");
    return &person.name;
}

fn main() {
    let mut person = Person { name: String::from("Ein Name"), };

    let name_referenz = &mut person.name;
    *name_referenz = String::from("Neuer Name");

    let returned_name = return_person_name(&mut person);

    println!("returned_name {} ", returned_name);
}
```

Ausgabe:

```
returned_name Neuer Name d
```

Achten Sie genau auf die Änderungen zum vorherigen Programm. Es wurde eine Lifetime an mehreren Stellen hinzugefügt und mut auch. TODO: Lifetime erklären.

36 Beispiel Referenz in Funktion ändern (Lifetime):

```
use std::ops::Add;

struct Person {
    name: String,
}

fn return_person_name<'a, 'x>(person_0p: &'a mut Person, person_1p: &'x mut Person) -> &'a String {
    person_0p.name = String::from (&person_0p.name).add(" 0d");
    person_1p.name = String::from (&person_1p.name).add(" 1d");
    return &person_0p.name;
}

fn main() {
    let mut person_0 = Person { name: String::from("Person nummer 0"), };
    let mut person_1 = Person { name: String::from("Person nummer 1"), };

    let returned_name = return_person_name(&mut person_0, &mut person_1);

    println!("returned_name {} ", returned_name);
    println!("person_0 {} ", person_0.name);
    println!("person_1 {} ", person_1.name);
}
```

Ausgabe:

```
returned_name Person nummer 0 0d
person_0 Person nummer 0 0d
person_1 Person nummer 1 1d
```

Die Personen lassen sich nun über eine Funktion ändern.

37 Beispiel Referenz über Funktion ändern:

```
use std::ops::Add;

struct Person {
    name: String,
}

fn change_person_name(person: &mut Person) {
    person.name = String::from(person.name.as_str()).add(" x");
}

fn main() {
    let mut person = Person { name: String::from("Ein Name"), };

    change_person_name(&mut person);
    change_person_name(&mut person);
    change_person_name(&mut person);

    println!("person {} ", person.name);
}
```

Ausgabe:

```
person Ein Name x x x
```

Innerhalb der Funktion wird die Referenz von `person.name` geholt und um ein „x“ erweitert. Da das ganze 3 Mal geschieht, haben wir zum Schluss 3 x hinter dem Namen stehen.

38 Beispiel Variable mit Referenz:

```
fn main() {  
  
    let mut zahl = 9;  
    let zahl_referenz = &mut zahl;  
  
    let kopie = *zahl_referenz; // kopiert jetzt die 9 in kopie  
  
    *zahl_referenz = 8;  
  
    //println!("{}", zahl); // es laesst sich entweder diese Zeile oder die darunter verwenden  
    println!("Referenz {}", zahl_referenz); // zahl und zahl_referenz koennen nicht beide verwendet werden  
    println!("Kopie {}", kopie);  
}
```

Ausgabe:

```
Referenz 8  
Kopie 9
```

Lassen Sie sich nicht beirren, vielleicht ist es an dieser Stelle noch nicht sinnvoll ein solches Beispiel zu bringen. Jedoch sollten Sie ein Gefühl dafür entwickeln wie man mit Kopien arbeiten kann. Der *-Operator ermöglicht einem eine Kopie des Wertes des Eigentümers zu erhalten.

39 Beispiel Variable : Angabe des Strings explizit mit &str

```
fn main() {  
    let mut meine_variable : &str = "Text des Strings";  
    println!("meine_variable: {} ", meine_variable );  
}
```

Ausgabe:

meine_variable: Text des Strings

Sollten Sie Umlaute (ü, ä, ö) verwenden, erhalten Sie ielleicht eine Compile-Fehler. ß jedoch funktioniert

40 Beispiel &str :

```
fn string_uebergabe(s:&str){  
    println!("in funktion {}",s);  
}  
  
fn main() {  
    string_uebergabe(&String::from("hier machen wir aus einem String ein &str"));  
    string_uebergabe("Dies ist ein &str");  
}
```

Ausgabe:

in funktion hier machen wir aus einem String ein &str
in funktion Dies ist ein &str

41 Beispiel Konvertierung:

```
fn string_to_integer_parse(s:&str)-> Result<(), std::num::ParseIntError> {
    let mein_parse = s.parse::<i32>()?;
    println!("mein_string parse in funktion string_to_integer {}", mein_parse);

    Ok(())
}

fn string_to_integer(s:&str)-> Result<(), std::num::ParseIntError> {
    let mein_string = s.to_string();
    println!("mein_string in funktion string_to_integer{}", mein_string);

    Ok(())
}

fn integer_to_string(mein_integer:i64)-> Result<(), std::num::ParseIntError> {
    let mein_string = mein_integer.to_string();
    println!("mein_string in funktion {}", mein_string);
    Ok(())
}

fn main() {

    let u:&str = "11";
    string_to_integer_parse(&u);

    let res = integer_to_string(33);
    println!("res in main {}", res.is_ok());

    let u:&str = "22";
    string_to_integer(&u);
}
```

```
mein_string parse in funktion string_to_integer 11
mein_string in funktion 33
res in main true
mein_string in funktion string_to_integer22
```

Wollen Sie Integer oder Floating zu Strings umwandeln, oder Strings in Zahlen, dann können Sie so vorgehen.

IF

Es gibt einige Operatoren und Kombinationen in If-Abfragen. Hauptsächlich kommen

```
!a // nicht
a > b //grösser
a < b // kleiner
a ==b // ist gleich
a != b // ist ungleich
a <= b // kleiner oder gleich
a >=b // grösser oder gleich
a && b // a und b müssen true sein
a||b // Entweder a oder b muss true sein.
```

```
if e >= 20 && e < 22
```

42 Beispiel if-Bedingung :

```
fn main() {

    let q = 28;

    if q == 1 {
        println!("q: {} ", q );
    }

}
```

Ausgabe:

Es wird nichts ausgegeben, da q nicht 1 ist.

43 Beispiel if-Bedingung :

```
fn main() {  
    let q = 28;  
  
    if q == 28  
    {  
        println!("q: {} ", q);  
    }  
}
```

Ausgabe:

q: 28

Es wird 28 ausgegeben.

44 Beispiel if-else :

```
fn main() {  
  
    let q = 28;  
  
    if q == 280 {  
        println!("if Block: {} ", q );  
    } else {  
        println!("else Block: {} ", q );  
    }  
  
}
```

Ausgabe:

```
else Block: 28
```

45 Beispiel if-else :

```
fn main() {  
  
    let q = 28;  
  
    println!("Ausgabe 1 sieht man immer");  
  
    if q == 280 {  
        println!("if Block: {} ", q);  
    } else {  
        println!("else Block: {} ", q);  
    }  
  
    println!("Ausgabe 2 sieht man auch immer");  
}
```

Ausgabe:

```
Ausgabe 1 sieht man immer  
else Block: 28  
Ausgabe 2 sieht man auch immer
```

46 Beispiel if - elseif - else :

```
fn main() {  
  
    let q = 28;  
  
    println!("Ausgabe 1 sieht man immer");  
  
    if q == 280 {  
        println!("if Block ß: {} ", q);  
    } else if q == 28 {  
        println!("else if 1. Block: {} ", q);  
    } else if q == 19 {  
        println!("else if 2. Block: {} ", q);  
    } else {  
        println!("else Block: {} ", q);  
    }  
  
    println!("Ausgabe 2 sieht man auch immer");  
}
```

Ausgabe:

```
Ausgabe 1 sieht man immer  
else if 1. Block: 28  
Ausgabe 2 sieht man auch immer
```

Der letzte else-Block könnte auch weggelassen werden.

Es muss immer mindestens ein if geben und es kann unendlich viele else-if geben. Else-Block kann, muss es aber nicht geben. Es gibt immer nur ein Else-Block zu einem if (also der der nur „else“ heisst). Man kann allerdings viele if-Abfragen untereinander machen.

47 Beispiel if - elseif - else und ein zweiter if-Block in der Funktion:

```
fn main() {  
  
    let q = 28;  
  
    println!("Ausgabe 1 sieht man immer");  
  
    if q == 280 {  
        println!("if Block: {} ", q);  
    } else if q == 28 {  
        println!("else if 1. Block: {} ", q);  
    } else if q == 19 {  
        println!("else if 2. Block: {} ", q);  
    } else {  
        println!("else Block: {} ", q);  
    }  
  
    println!("Ausgabe 2 sieht man auch immer");  
  
    let h = "mehrere Buchstaben";  
    if h == "irgendein Buchstabe" {  
        println!("if Block 2: {} ", q);  
    }  
}
```

Ausgabe:

```
Ausgabe 1 sieht man immer  
else if 1. Block: 28  
Ausgabe 2 sieht man auch immer
```

Der Inhalt des 2te if-Blocks wird nicht ausgeführt.

48 Beispiel bool:

```
fn main() {  
  
    let g:bool = true;  
  
    if g == true {  
        println!("g: {}", g);  
    }  
}
```

Ausgabe:

```
g : true
```

Ein bool (oder boolean) speichert nur true oder false. Die obige if-Abfrage wird von vielen Entwicklern nicht so gerne gesehen.

49 Beispiel bool:

```
fn main() {  
  
    let g:bool = true;  
  
    if g {  
        println!("g : {}", g);  
    }  
}
```

Ausgabe:

```
g : true
```

In diesem Beispiel fehlt das `== true`, das liegt daran, dass wenn kein `==true` dasteht, ist das so, als ob man `g == true` geschrieben hätte.

50 Beispiel bool:

```
fn main() {  
  
    let g:bool = false;  
  
    if !g {  
        println!("g : {}", g);  
    }  
}
```

Ausgabe:

```
g : false
```

Hier fragen wir ab, ob das bool false ist. Wir hätten auch nach `g == false` fragen können.

51 Beispiel while:

```
fn main() {  
  
    let mut a = 255;  
  
    while a > 19 {  
        println!("a ist groesser als 19 a:{}", a);  
        a = a - 30;  
    }  
}
```

Ausgabe:

```
a ist groesser als 19 a:255  
a ist groesser als 19 a:225  
a ist groesser als 19 a:195  
a ist groesser als 19 a:165  
a ist groesser als 19 a:135  
a ist groesser als 19 a:105  
a ist groesser als 19 a:75  
a ist groesser als 19 a:45
```

Zuerst ist a = 255 , danach 225etc...

52 Beispiel while:

```
fn main() {  
  
    let mut a = 255;  
  
    while a > 19 {  
        println!("a ist groesser als 19 a:{}", a);  
        a = a - 30;  
        println!("a:{}", a);  
    }  
}
```

Ausgabe:

```
a ist groesser als 19 a:255  
a:225  
a ist groesser als 19 a:225  
a:195  
a ist groesser als 19 a:195  
a:165  
a ist groesser als 19 a:165  
a:135  
a ist groesser als 19 a:135  
a:105  
a ist groesser als 19 a:105  
a:75  
a ist groesser als 19 a:75  
a:45  
a ist groesser als 19 a:45  
a:15
```

Jetzt sehen wir in diesem Beispiel, dass nach dem zurücksetzen von a um 30, ist a tatsächlich 30 weniger.

53 Beispiel loop:

```
fn main() {  
  
    let mut a = 255;  
  
    loop {  
        println!("wird unendlich ausgeführt");  
    }  
}
```

Ausgabe:

```
wird unendlich ausgeführt  
wird unendlich ausgeführt  
wird unendlich ausgeführt  
....
```

Diese Schleife wird erst beendet, wenn der Prozess beendet wird.

54 Beispiel loop mit Rückgabewert:

```
fn main() {  
  
    let mut i = 100;  
  
    let loop_ergebnis = loop {  
        i = i + 20; // oder i += 20  
        if i == 200 {  
            //oder: break "ein String kann auch zurueckgegeben werden";  
            break 3.33f32;  
        }  
    };  
  
    println!("loop_ergebnis: {}", loop_ergebnis);  
}
```

Ausgabe:

```
loop_ergebnis: 3.33
```

Beachten Sie dass i mutable sein muss, was Ihnen der Compiler aber eh sagen wird.

55 Beispiel for:

```
fn main() {  
  
    //for-Schleife  
  
    for t in 8..19 {  
        println!("t: {}",t);  
    }  
}
```

Ausgabe:

```
t: 8  
t: 9  
t: 10  
t: 11  
t: 12  
t: 13  
t: 14  
t: 15  
t: 16  
t: 17  
t: 18
```

Diese Schleife t erhält zunächst den Wert 8. Mit den Wert 19 wird die Schleife jedoch nicht ausgegeben. D.h. solange t < 19 ist wird die Schleife ausgeführt.

Das ganze bedeutet soviel wie

```
let t = 8
```

```
while t < 19 {}
```

56 Beispiel for-enumerate:

```
fn main() {  
  
    //for-enumerate-Schleife  
    for (schritt, zahl_aus_enumerate) in (8..13).enumerate(){  
        println!("schritt: {}, zahl_aus_enumerate : {}",schritt,zahl_aus_enumerate);  
    }  
}
```

Ausgabe:

```
schritt: 0, zahl_aus_enumerate : 8  
schritt: 1, zahl_aus_enumerate : 9  
schritt: 2, zahl_aus_enumerate : 10  
schritt: 3, zahl_aus_enumerate : 11  
schritt: 4, zahl_aus_enumerate : 12
```

Die erste Variable ist einfach ein Zähler, der bei 0 beginnt, die zweite Zahl ist dann die Zahl 8-12 (also < 13).

57 Beispiel for-Schleife verschachtelt:

```
fn main() {  
  
    for x in 0..4 {  
        for y in 3..5 {  
            println!("x {}, y {}", x, y);  
        }  
    }  
}
```

Ausgabe:

```
x 0, y 3  
x 0, y 4  
x 1, y 3  
x 1, y 4  
x 2, y 3  
x 2, y 4  
x 3, y 3  
x 3, y 4
```

58 Beispiel for-Schleife break:

```
fn main() {  
  
    for t in 8..19 {  
        println!("t1: {}",t);  
        if t == 11 {  
            break;  
        }  
        println!("t2: {}",t);  
        println!(" ");  
    }  
    println!("Ende");  
}
```

Ausgabe:

```
t1: 8  
t2: 8  
  
t1: 9  
t2: 9  
  
t1: 10  
t2: 10  
  
t1: 11  
Ende
```

Mit dem Schlüsselwort break kann man eine for-Schleife vorzeitig beenden.

59 Beispiel for-Schleife continue:

```
fn main() {  
  
    for t in 14..19 {  
        println!("t1: {}",t);  
        if t < 17 {  
            continue;  
        }  
        println!("t2: {}",t);  
        println!(" ");  
    }  
    println!("Ende");  
}
```

Ausgabe:

```
t1: 14  
t1: 15  
t1: 16  
t1: 17  
t2: 17  
  
t1: 18  
t2: 18  
  
Ende
```

Mit dem Schlüsselwort `continue` kann man eine `for`-Schleife dazu bringen den unteren Teil (also unter `continue` auszulassen) und die `for`-Schleife mit dem nächsten Wert wieder von oben weiterzuführen.

`t` ist nur innerhalb der `for`-Schleife verwendbar.

60 Beispiel for-Schleife continue mit Labeln:

```
fn main() {  
  
    'aeussere: for x in 0..4 {  
        'innere: for y in 3..5 {  
            println!("x {}, y {}", x, y);  
            if x == 2 {  
                continue 'aeussere;  
            }  
        }  
    }  
}
```

Ausgabe:

```
x 0, y 3  
x 0, y 4  
x 1, y 3  
x 1, y 4  
x 2, y 3  
x 3, y 3  
x 3, y 4
```

Die Namen der Label lassen sich beliebig vergeben. Das Schlüsselwort break kann man auch verwenden. In diesem Beispiel wird das Label „innere“ nicht verwendet.

61 Beispiele für for-Schleifen verschiedener Arten:

```
fn main() {  
  for e in 2..6 {  
    println!(" 2..6 :{}", e);  
  }  
  
  println!(" ");  
  
  for e in 2..=6 {  
    println!(" 2..=6 :{}", e);  
  }  
  
  println!();  
  
  for h in (0..5).step_by(2) {  
    println!("h {}", h);  
  }  
  
  println!();  
  
  for h06 in (0..6).step_by(2) {  
    println!("h06 {}", h06);  
  }  
  
  println!();  
  
  for h07 in (0..7).step_by(2) {  
    println!("h07 {}", h07);  
  }  
  
  println!();  
  
  for u in (-10..10).map(|x| x as f64 * 0.1) {  
    println!("u {}", u);  
  }  
}
```

Ausgabe:

```
2..6 :2  
2..6 :3  
2..6 :4  
2..6 :5  
  
2..=6 :2  
2..=6 :3  
2..=6 :4  
2..=6 :5  
2..=6 :6  
  
h 0  
h 2  
h 4
```

```
h06 0
h06 2
h06 4

h07 0
h07 2
h07 4
h07 6

u -1
u -0.9
u -0.8
u -0.7000000000000001
u -0.6000000000000001
u -0.5
u -0.4
u -0.30000000000000004
u -0.2
u -0.1
u 0
u 0.1
u 0.2
u 0.30000000000000004
u 0.4
u 0.5
u 0.6000000000000001
u 0.7000000000000001
u 0.8
u 0.9
```

Beachten Sie, daß man auch ein `step_by` mit übergeben kann, also eine Schrittweite. Wenn Sie durch floating Werte druchiterieren wollen, sollten Sie dies wie in dem Beispiel mit der `Map`-Anweisung tun.

62 Beispiel Gültigkeitsbereich einer Variablen (oder Konstanten):

```
fn main() {  
  
    let g = 88;  
  
    let mut x = 2;  
    if x == 2 {  
        // g ist jetzt nur innerhalb des if-Blocks gültig  
        // und in verschachtelten Blocks  
        let g = 8;  
        println!("g 1.Ausgabe: {}", g);  
    }  
  
    // g ist hier g=88  
    println!("g 2.Ausgabe: {}", g);  
}
```

Ausgabe:

```
g 1.Ausgabe: 8  
g 2.Ausgabe: 88
```

In Java wäre es nicht möglich eine Variablennamen innerhalb eines Blocks doppelt zu vergeben.

Aufgabe:

Was passiert wenn man folgenden Code ausführt?:

```
fn main() {  
    let a = 999;  
    {  
        let a = 9999;  
        println!("a 4.Ausgabe: {}", a);  
    }  
}
```

Aufgabe:

Was passiert wenn man folgenden Code ausführt? Kompiliert der Code? Kann er ausgeführt werden?:

```
fn main() {  
    let a = 999;  
    let a = 9;  
    {  
        let a = 9999;  
        println!("a 4.Ausgabe: {}", a);  
    }  
}
```

Aufgabe:

Was passiert wenn man folgenden Code ausführt? Kompiliert der Code? Kann er ausgeführt werden?:

```
fn main() {
```

```
let mut a = 999;  
let a = 9;  
println!("a 4.Ausgabe: {}", a);  
}
```

63 Beispiel eines Blocks mit Rückgabewert:

```
fn main(){  
  let rueckgabe_wert = {  
    println!("hier tun wir etwas sinnvolles...");  
    let mein_string = String::from("mein String");  
    mein_string + " und ein anderer String"  
  };  
  println!("rueckgabe_wert: {}", rueckgabe_wert);  
}
```

Ausgabe:

```
rueckgabe_wert: mein String und ein anderer String
```

Hier sehen Sie, dass ein Block ein Rückgabewert enthalten kann, dies kent man so z.B. aus Java nicht (Stand: 20.6.2022).

64 Beispiel bool:

```
fn main() {  
    let g:bool = false;  
  
    if !g {  
        println!("g : {}", g);  
    }  
}
```

Ausgabe:

```
g : false
```

Der Variablentyp (auch für Konstanten) bool kann nur true oder false haben. In Java oder C++ z.B. gibt es ihn auch. In Java gibt es auch einen Boolean (also mit einem grossen B beginnend) der 3 Zustände speichern kann, nämlich true, false oder null.

65 Beispiel bool &&:

```
fn main() {  
  
    let g:bool = false;  
    let z:bool = true;  
  
    if !g && z {  
        println!("g 1: {}", g);  
        println!("z 1: {}", z);  
    }  
    println!("g 2: {}", g);  
    println!("z 2: {}", z);  
}
```

Ausgabe:

```
g 1: false  
z 1: true  
g 2: false  
z 2: true
```

`g && z` bedeutet wenn `g` und `z` (also beide auf `true` stehen, dann gehe in den `if`-Block).

`g || z` bedeutet wenn `g` oder `z` (also einer von beiden `true` stehen, dann gehe in den `if`-Block).

66 Beispiel bool &&:

```
fn main() {  
  
    let g:bool = false;  
    let z:bool = true;  
  
    if !(g && z) {  
        println!("g 1: {}", g);  
        println!("z 1: {}", z);  
    }  
    println!("g 2: {}", g);  
    println!("z 2: {}", z);  
}
```

Ausgabe:

```
g 1: false  
z 1: true  
g 2: false  
z 2: true
```

Sie können runde Klammern setzen um bestimmte Variablen zu verneinen.

67 Beispiel bool:

```
fn main() {  
  
    let g:bool = false;  
    let z:bool = true;  
  
    if (g && z) {  
        println!("g 1: {}", g);  
        println!("z 1: {}", z);  
    }  
    println!("g 2: {}", g);  
    println!("z 2: {}", z);  
}
```

Ausgabe:

```
g 2: false  
z 2: true
```

Sie können auch runde Klammern setzen auch wenn es keine Sinn macht.

68 Beispiel Character:

```
fn main() {  
    let r:char = 'f';  
    println!("r : {}", r);  
}
```

Ausgabe:

```
r : f
```

Character sind einzelne Buchstaben, also immer nur ein Zeichen. In den meisten Anwendungen in denen ich mitgearbeitet habe haben sie praktisch keine Rolle gespielt. Meisten wurden Strings benutzt.

69 Beispiel Character:

```
fn main(){  
    let text = String::from("Q*bert");  
    for x in text.chars() {  
        println!("Character: {}", x);  
    }  
}
```

Ausgabe:

```
Character: Q  
Character: *  
Character: b  
Character: e  
Character: r  
Character: t
```

Eine Möglichkeit die einzelnen Character aus einem String darzustellen.

70 Beispiel Character:

```
fn main(){
    //let mein_text = "mein String Inhalt";
    let mein_text = "дптв";
    let laenge_mein_text = mein_text.len()+1;

    println!("laenge_mein_text+1: {}", laenge_mein_text);

    for x in (2..laenge_mein_text).step_by(2){
        let s = &mein_text[x-2..x];
        println!("s: {}", s);
    }
}
```

Ausgabe:

```
laenge_mein_text+1: 9
s: д
s: п
s: т
s: в
```

Versuchen Sie den ersten `mein_text` mal auszugeben. Sie werden sehen, dass es vllt. Nicht ganz das ist was Sie erwarten. Sie können mit Rust nicht davon ausgehen, dass Character einfach immer dieselbe Grösse haben.

71 Beispiel Integer:

```
fn main() {  
    let r: i8 = 64;  
    let r2: i32 = 2147000000;  
  
    println!("r : {}", r);  
    println!("r2 : {}", r2);  
}
```

Ausgabe:

```
r : 64  
r2 : 2147000000
```

Integer sind ganze Zahlen i8 z.B. geht von -128 bis +127.

72 Beispiel Integer und const:

```
fn main() {  
  
    pub const MY_MAX: i8 = i8::MAX; // 127i8  
    pub const MY_MIN: i8 = i8::MIN; // -128i8  
  
    let r: i8 = 64;  
    let r2: i32 = 2147000000;  
  
    println!("r : {}", r);  
    println!("r2 : {}", r2);  
  
    println!("MY_MAX : {}", MY_MAX);  
    println!("MY_MIN : {}", MY_MIN);  
}
```

Ausgabe:

```
r : 64  
r2 : 2147000000  
MY_MAX : 127  
MY_MIN : -128
```

Es ist einfach in RUST den maximalen und minimalen Integer Wert zu lesen. Im Gegensatz zu Java oder C++ kann nicht einfach +1 auf den Maximal Wert oder -1 auf den Minimalwert draufgerechnet werden. Konstanten sollten immer mit SCREAMING_SNAKE_CASE also grossem Snake_Case geschrieben werden.

73 Beispiel Integer:

```
fn main() {  
    let z: i32;  
    z = i32::min_value();  
    println!("z : {}", z);  
}
```

Ausgabe:

```
z : -2147483648
```

Auch so war es möglich an den minimalen Wert eines i32 zu kommen.

74 Beispiel Integer (unsigned):

```
fn main() {  
    let z1: u32;  
    let z2: u32;  
    z1 = u32::min_value();  
    z2 = u32::MAX;  
    println!("z1 : {}", z1);  
    println!("z2 : {}", z2);  
}
```

Ausgabe:

```
z1 : 0  
z2 : 4294967295
```

mit u (unsigned) gibt man an, dass sich ein Integer nur im positiven Bereich bewegen soll.

75 Beispiel Float :

```
fn main() {
    let z1: f32;
    let z2: f32;
    z1 = f32::MIN;
    z2 = f32::MAX;
    println!("z1 : {}", z1);
    println!("z2 : {}", z2);
}
```

Ausgabe:

[illegible]

76 Beispiel Float :

```
fn main() {  
    let z1: f32;  
    let z2: f32;  
    z1 = 3.454545454522222;  
    z2 = -232344.232323232323234;  
    println!("z1 : {}", z1);  
    println!("z2 : {}", z2);  
}
```

Ausgabe:

```
z1 : 3.4545455  
z2 : -232344.23
```

Bei Fließkommazahlen muss man mit der Ausgabe mittels println aufpassen, es werden einfach ein paar Stellen abgeschnitten.

77 Beispiel Float Konvertierung:

```
fn main() {  
    let floating_f64:f64 = 2.0;  
  
    let floating_f32 = 55.55f32;  
  
    //not correct: try it: it says: ^^^^^^^^^^^ expected `f64`, found `f32`  
    //let sum_of_different_floats_incorrect = floating_f64 + floating_f32;  
    //println!("sum_of_different_floats_incorrect {}", sum_of_different_floats_incorrect);  
  
    //not correct: try it: it says: ^^^^^^^^^^^ expected `f64`, found `f32`  
    //let sum_of_different_floats = floating_f64 as f64 + floating_f32;  
    //println!("sum_of_different_floats {}", sum_of_different_floats);  
  
    //correct:  
    let sum_of_different_float_types = floating_f64 + floating_f32 as f64;  
    println!("sum_of_different_float_types {}", sum_of_different_float_types);  
}
```

Ausgabe:

```
sum_of_different_float_types 57.54999923706055
```

Fließkommazahlen die nicht vom selben Typ sind, also z.B. f32 und f64 müssen explizit bei der Summierung mitangegeben werden, wobei dem kleineren Typ gesagt werden muss, dass er vom Typ des größeren sein soll.

78 Beispiel Float und Integer Summierung:

```
fn main() {  
    let floating_point_number:f64 = 1222.0;  
  
    let my_integer_8: i8 = 55;  
  
    let sum_of_different_types = floating_point_number + my_integer_8 as f64;  
    println!("sum_of_different_types {}", sum_of_different_types);  
}
```

Ausgabe:

```
sum_of_different_types: 1277
```

Fließkommazahl und Integer müssen explizit bei der Summierung der Typ mitangegeben werden. Würde man es nicht tun, dann würde die Fehlermeldung „^ no implementation for `f64 + i8`“ beim Kompilieren kommen.

79 Beispiel Boolean zu Integer Konvertierung:

```
fn main() {  
    let my_bool = false;  
    println!("{}", my_bool as i32);  
  
    let my_bool = true;  
    println!("{}", my_bool as i8);  
}
```

Ausgabe:

```
0  
1
```

80 Beispiel Array :

```
fn main() {  
    let z: [i32;4] = [3, 45, 234, 55];  
    println!("z[0] : {}", z[0]);  
    println!("z[1] : {}", z[1]);  
    println!("z[2] : {}", z[2]);  
    println!("z[3] : {}", z[3]);  
}
```

Ausgabe:

```
z[0] : 3  
z[1] : 45  
z[2] : 234  
z[3] : 55
```

Ein Array ist eine Ansammlung eines bestimmten Typs. Also Mehrere Elemente unter einem Variablennamen z.B. vom Typ i32. In diesem Fall geben wir an, dass es sich um ein Array vom Typ i32 mit 4 Elementen handelt. Die Angabe : `[i32;4]` hätte man aber bei Rust auch weglassen können.

Bei Arrays wird mit der Zahl 0 angefangen zu zählen. Das sechste Element ist als das an der Stelle 5. `x[5]` ist das 6te Element im Array weil `x[0]`, `x[1]`, etc...

81 Beispiel Array ohne Angabe des Typs und Anzahl der Elemente:

```
fn main() {  
    let z = [3, 45, 234, 55];  
    println!("z[0] : {}", z[0]);  
    println!("z[1] : {}", z[1]);  
    println!("z[2] : {}", z[2]);  
    println!("z[3] : {}", z[3]);  
}
```

Ausgabe:

```
z[0] : 3  
z[1] : 45  
z[2] : 234  
z[3] : 55
```

Rust macht in diesem Fall automatisch ein Array vom Typ `i32` mit 4 Elementen daraus. Später lassen sich keine neue Elemente mehr hinzufügen (also davor oder danach).

82 Beispiel Array :

```
fn main() {  
    let mut z = [3, 45, 234, 55];  
    println!("z[0] : {}", z[0]);  
    println!("z[1] : {}", z[1]);  
    println!("z[2] : {}", z[2]);  
    println!("z[3] : {}", z[3]);  
  
    z[2]=33;  
    println!("z[2] : {}", z[2]);  
}
```

Ausgabe:

```
z[0] : 3  
z[1] : 45  
z[2] : 234  
z[3] : 55  
z[2] : 33
```

Man kann aber die Elemente austauschen, wenn das Array mit „mut“ veränderbar gemacht wurde.

Aufgabe:

Probieren Sie aus, was passiert, wenn Sie z.B. einem Array mit 4 Elementen an der Stelle 6 als z[6] (der siebte Wert also) einen Wert hinzufügen.

Lösung:

?

→ Index out of bounds

83 Beispiel Array (4 Elemente mit Zahl 3 vorbelegen) :

```
fn main() {  
  let z= [3; 4];  
  println!("z[0] : {}", z[0]);  
  println!("z[1] : {}", z[1]);  
  println!("z[2] : {}", z[2]);  
  println!("z[3] : {}", z[3]);  
}
```

Ausgabe:

```
z[0] : 3  
z[1] : 3  
z[2] : 3  
z[3] : 3
```

Wenn man ein Array mit einer bestimmten Anzahl von Elementen mit einer bestimmten Zahl vorbelegen möchte, dann kann man das wie oben tun. Beachten Sie das Semikolon.

84 Beispiel Array → Anzahl Elemente des Arrays ermitteln :

```
fn main() {  
    let z= [3,6,5,1];  
    println!("z[0] : {}", z[0]);  
    println!("z[1] : {}", z[1]);  
    println!("z[2] : {}", z[2]);  
    println!("z[3] : {}", z[3]);  
  
    println!("z.len() : {}", z.len());  
}
```

Ausgabe:

```
z[0] : 3  
z[1] : 6  
z[2] : 5  
z[3] : 1  
z.len() : 4
```

Will man die Anzahl der Elemente im Programm verwenden, dann kann man das mittels „Variablenname.len()“ tun.

85 Beispiel Array in for-Schleife ausgeben :

```
fn main() {  
    let z=[3,6,5,1];  
  
    for g in &z {  
        println!("g : {}", g);  
    }  
}
```

Ausgabe:

```
g : 3  
g : 6  
g : 5  
g : 1
```

Um alle Elemente eines Arrays in einer for-Schleife ausgeben zu können, kann man die mit einem &-Operator machen.

86 Beispiel Slice (Referenz auf eine Variable) :

```
fn main() {  
    let z=[0,1,2,3,44,55,66,77,88];  
  
    for g in &z[3..7] { // liefert die 4te bis 8te Stelle, also Element an der Stelle 3 und 7.  
        println!("g : {}", g);  
    }  
}
```

Ausgabe:

```
g : 3  
g : 44  
g : 55  
g : 66
```

Ein Slice zeigt auf eine feste Stelle.

Mit &g[..] kann man alle Element im Array ansprechen.

&r[2..5] liefert alle Elemente von Stelle 2 bis 5.

Im Beispiel oben wird die 4te bis 8te Stelle geliefert, also Element 3 bis 7 (da wir mit 0 anfangen zu zählen).

87 Beispiel String Slice :

```
fn main() {  
    let mein_string = "Dies ist irgendein Text";  
  
    println!("mein_string .is_empty {}", mein_string.is_empty());  
  
    println!("mein_string &s[1..=8] {}", &mein_string[1..=8]);  
    println!("mein_string &s[1..=8] {}", &mein_string[1..8]);  
    // println!("mein_string &s[1..=8] {}", &mein_string[1..88]);  
  
    println!("mein_string .starts_with {}", mein_string.starts_with("d"));  
    println!("mein_string .starts_with {}", mein_string.starts_with("D"));  
  
    println!("mein_string .ends_with {}", mein_string.ends_with("t"));  
    println!("mein_string .ends_with {}", mein_string.ends_with("T"));  
  
    println!("mein_string .len {}", mein_string.len());  
  
    println!("mein_string .find {}", mein_string.find("irgen").unwrap());  
    println!("mein_string .find {}", mein_string.find("Irgen").unwrap());  
}
```

Ausgabe:

```
mein_string .is_empty false  
mein_string &s[1..=8] ies ist  
mein_string &s[1..=8] ies ist  
mein_string .starts_with false  
mein_string .starts_with true  
mein_string .ends_with true  
mein_string .ends_with false  
mein_string .len 23  
mein_string .find 9
```

Schauen Sie auch, was passiert, wenn die auskommentierten Zeilen einkommentiert werden.

88 Beispiel Tupel :

```
fn main() {  
  
    let mut mein_tupel: (&str, f32, i32) = ("Eine Zeichenkette" , 2.35, 6);  
  
    let(x,y,z) = mein_tupel;  
  
    println!("x : {}" , x);  
    println!("y : {}" , y);  
    println!("z : {}" , z);  
}
```

Ausgabe:

```
x : Eine Zeichenkette  
y : 2.35  
z : 6
```

Mit Tupeln können z.B. Funktionen mehr als einen Rückgabewert zurückliefern. (Tupel erinnern ein wenig an den Datentyp Vector in Java, der kann auch Elemente von verschiedenen Typen in sich ablegen, bzw. referenzieren).

Möchte man ein Tupel mit nur einem Element anlegen dann darf das nicht so

let v = (5);

gemacht werden, sondern man muss

let v = (5,)

schreiben.

Es lassen sich auch Arrays als Elemente in Tupeln speichern.

89 Beispiel Tupel :

```
fn main() {  
  
    let mut mein_tupel: (&str, f32, i32) = ("Eine Zeichenkette" , 2.35, 6);  
  
    let(x,y,z) = mein_tupel;  
  
    println!("x : {}" , mein_tupel.0);  
    println!("y : {}" , mein_tupel.1);  
    println!("z : {}" , mein_tupel.2);  
}
```

Ausgabe:

```
x : Eine Zeichenkette  
y : 2.35  
z : 6
```

Mit dem Punkt Operator können wir die einzelnen Stellen im Tupel direkt ansprechen.

Aufgabe:

Probieren Sie aus, was passiert, wenn Sie `mein_tupel.8` ansprechen wollen.

90 Beispiel Funktion mit Tupeln:

```
fn main() {  
  
    let (z1, z2, z3) = zahlen_mal_zwei(3,4,5);  
    println!("z1: {}, z2: {}, z3:{}", z1, z2, z3);  
  
    let mein_tupel = zahlen_mal_zwei(1, 3, 12);  
    println!("mein_tupel.0: {}, mein_tupel.1: {}, mein_tupel.2: {}", mein_tupel.0, mein_tupel.1, mein_tupel.2);  
}  
  
fn zahlen_mal_zwei(a: i8, b: i8, c:i8) -> (i8, i8, i8) {  
    return (a*2, b*2, c*2);  
}
```

Ausgabe:

```
z1: 6, z2: 8, z3:10  
mein_tupel.0: 2, mein_tupel.1: 6, mein_tupel.2: 24
```

Tupel (tuple) können wie bereits erwähnt mehrere verschiedene Typen haben, müssen sie aber nicht. Man kann sie als eine Variable oder als mehrer wieder entgegennehmen.

Es lassen sich auch leere Tupel zurückgeben, nur der Information halber:

```
fn return_empty_tuple_() {  
}  
  
fn return_empty_tuple() -> () {  
    return ();  
}
```

91 Beispiel Tupel in Funktion übergeben:

```
fn main() {  
    let laenge_breite_hoehe : (i64, i64, i64) = (4i64, 3i64,2);  
  
    let ergebnis = volumen_berechnung(laenge_breite_hoehe);  
    println!("Volumen= {}",ergebnis);  
}  
  
fn volumen_berechnung(volumen: (i64, i64, i64)) -> i64 {  
    volumen.0 * volumen.1 *volumen.2  
}
```

Ausgabe:

```
Volumen= 24
```

In diesem Beispiel sehen wir, wie man ein Tupel in einer Funktion als Parameter übergeben, bzw. entgegennehmen kann.

92 Beispiel Tupel als struct:

```
struct Mein_Tupel<'a>(&'a str, f64);

fn main() {
    let instanz_von = Mein_Tupel("text_", 32.01 );

    println!("0: {}, 1: {}", instanz_von.0, instanz_von.1);
}
```

Ausgabe:

```
0: text_, 1: 32.01
```

Beachten Sie hier die Lifetime Angabe 'a beim String , hierauf wir später noch eingegangen. Würden wir ohne String arbeiten, könnte das Struct wie folgt aussehen. Ein Beispiel:

```
struct Mein_Tupel(i8, f64);

fn main() {
    let instanz_von = Mein_Tupel(12, 32.01 );

    println!("0: {}, 1: {}", instanz_von.0, instanz_von.1);
}
```

Ausgabe:

```
0: 12, 1: 32.01
```

93 Beispiel Funktion als Variable :

```
fn main() {  
    println!("vor Funktionsaufruf....");  
  
    let z = meine_funktion;  
    println!("...z...{}", z (4,6));  
}  
  
fn meine_funktion(a:i32, mut ypsilon:i32) -> i32 {  
    ypsilon+a  
}
```

Ausgabe:

```
vor Funktionsaufruf....  
...z...10
```

Achten Sie darauf, dass Sie die Variablen `z` keinen Variablentyp geben wie „`let z:i32`“. Wie das geht sehen Sie im nächsten Beispiel, wenn Sie es nicht weglassen wollen

94 Beispiel Funktion als Variable:

```
fn main() {  
  println!("vor Funktionsaufruf....");  
  
  let z:fn(i32,i32)-> i32 = meine_funktion;  
  println!("...z...{}", z (4,6));  
}  
  
fn meine_funktion(a:i32, mut ypsilon:i32) -> i32 {  
  ypsilon+a  
}
```

Ausgabe:

```
vor Funktionsaufruf....  
...z...10
```

Bei Funktionen als Variablen muss man einfach, sowas wie
let z: fn(i32, i32, i32) → i32
angeben.

Man kann die Funktion die aufgerufen wird hierdurch zur Laufzeit wechseln. Probieren Sie es aus.

95 Beispiel struct :

```
struct MeinStruct {  
    z: f32,  
    u: i32,  
  
    //mut darf hier nicht verwendet werden  
}  
  
fn main() {  
    let tt = MeinStruct {z: 3.4, u: 5 };  
    println!("...tt.u...{}", tt.u );  
    println!("...tt.z...{}", tt.z );  
}
```

Ausgabe:

```
...tt.u...5  
...tt.z...3.4
```

Mit Structs kann man ähnlich wie mit Klassen in Java neue Variablentypen bilden. Das Konzept werden viele aus C kennen.

Die Elemente in structs nennt man fields (in Java Instanzvariablen oder früher in C++ Membervariablen, bei Klassen, in C Member of a structure).

Ein Struct kann aber auch mehrere verschiedene fields mit unterschiedlichen Typen haben, z.B.:

```
struct Irgendeinstrut {  
    b:f32,  
    irgendein_text: String,  
    mein_vector: Vec<String>,  
    mein_vector_ints: Vec<i32>,  
}
```

96 Beispiel Struct :

```
struct MeinStruct {  
    z: f32,  
    u: i32,  
}  
  
fn main() {  
    // tt ist owner  
    let tt = MeinStruct {z: 3.4, u: 5 };  
    println!("...tt.u...{}", tt.u );  
    println!("...tt.z...{}", tt.z );  
  
    // t_neu ist owner  
    let mut t_neu = MeinStruct {z: 0.0, u: 0};  
    t_neu.u = 8;  
    println!("...t_neu.u...{}", t_neu.u );  
    println!("...t_neu.z...{}", t_neu.z );  
}
```

Ausgabe:

```
...tt.u...5  
...tt.z...3.4  
...t_neu.u...8  
...t_neu.z...0
```

Man kann die Variablen verändern, wenn man mut mitangibt.

tt und t_neu sind jeweils Owner der jeweiligen Instanzen der angelegten Structs. Sobald die Funktion beendet ist werden. Die Variablen und die Instanzen werden nach Beendigung der Funktion direkt gelöscht, die nennt man „drop“. Sollte eine Instanz Instanzen anderer Structs haben würde zuerst die obere Instanz und danach die Kind-Instanzen gelöscht („dropped“).

97 Beispiel Struct mit borrowed Instanz:

```
struct Konto {  
    guthaben: i64,  
}  
  
fn main() {  
    let konto1 = Konto { guthaben: 123_321_444 };  
    let konto_borrowed = &konto1; // & mal weglassen und ausprobieren  
  
    println!("Konto1: {}", konto1.guthaben);  
    println!("Konto_borrowed: {}", konto_borrowed.guthaben);  
}
```

Ausgabe:

```
Konto1: 123321444  
Konto_borrowed: 123321444
```

Mithilfe des &-Operators kann man den Zugriff auf eine andere Instanz ausleihen. Probieren Sie aus, was passiert wenn Sie das & weglassen.

98 Beispiel Eigentümerschaft:

```
struct Konto {  
    guthaben: f64,  
    kredit: f64,  
}  
  
fn main() {  
    let mut konto = Konto {  
        guthaben: 3000f64,  
        kredit: 500.0,  
    };  
  
    println!(  
        "Konto: {} ",  
        differenz(&konto) // ausprobieren & mal zu loeschen  
    );  
  
    konto.guthaben = 200.0; // ohne & : value partially assigned here after move  
}  
  
fn differenz(mkonto: &Konto) -> f64 { // ausprobieren & mal zu loeschen  
    mkonto.guthaben - mkonto.kredit  
}
```

Ausgabe:

```
Konto: 2500
```

Probieren Sie aus, was passiert, wenn Sie die & weglassen.

Sie werden dann sehen, dass wenn Sie die & wegmachen, dann können Sie `konto.guthaben` unter dem `println` nicht mehr ändern, wenn Sie die Methode aufrufen. Lassen Sie den Aufruf der Funktion weg, geht es trotzdem. Das liegt an der Eigentümerschaft. Diese können Sie mithilfe von & verlängern.

99 Beispiel Struct mit Verwendung in einer Funktion:

```
struct Konto {  
    guthaben: i64,  
}  
  
struct Person {  
    name: String,  
    konto: Konto,  
}  
  
fn irgendeine_Funktion (irgendeine_person: Person) {  
    println!("Person aus Funktion: {}", irgendeine_person.name );  
    //irgendeine_person.konto = Konto { guthaben: 123_321_444 };  
}  
  
fn main() {  
  
    let konto1 = Konto { guthaben: 123_321_444 };  
    let person1 = Person {name: String::from("Tibor"), konto: konto1 };  
  
    println!("Person: {}", person1.name );  
    println!("Konto von: {} hat Guthaben mit: {} ",person1.name, person1.konto.guthaben);  
  
    irgendeine_Funktion(person1);  
}
```

Ausgabe:

```
Person: Tibor  
Konto von: Tibor hat Guthaben mit: 123321444  
Person aus Funktion: Tibor
```

In diesem Beispiel sieht man wie ein Konto eine Instanz innerhalb einer Instanz einer Person ist. Nach der Beendigung der Funktion main wird erst die Person und dann das Konto gelöscht, gedropped. Versuchen Sie die auskommentierte Zeile innerhalb der Funktion „irgendeine_funktion“ mal auszukommentieren und zu schauen, ob das Programm noch kompiliert.

100 Beispiel Struct mit Verwendung in einer Funktion und mut:

```
struct Konto {  
    guthaben: i64,  
}  
  
struct Person {  
    name: String,  
    konto: Konto,  
}  
  
fn irgendeine_funktion (mut irgendeine_person: Person) {  
    println!("Person aus Funktion: {}", irgendeine_person.name );  
    irgendeine_person.konto = Konto { guthaben: 123_321_444 };  
}  
  
fn main() {  
  
    let konto1 = Konto { guthaben: 123_321_444 };  
    let person1 = Person {name: String::from("Tibor"), konto: konto1 };  
  
    println!("Person: {}", person1.name );  
    println!("Konto von: {} hat Guthaben mit: {} ",person1.name, person1.konto.guthaben);  
  
    irgendeine_funktion(person1);  
}
```

Ausgabe:

```
Person: Tibor  
Konto von: Tibor hat Guthaben mit: 123321444  
Person aus Funktion: Tibor
```

Um das Konto innerhalb der Funktion ändern zu können müssen wir jedoch erst den Parameter mit „mut“ angeben (mut = mutable = veränderbar) .

101 Beispiel Struct mit Rückgabe einer Instanz:

```
struct Konto {
    guthaben: i64,
}

struct Person {
    name: String,
    konto: Konto,
}

fn irgendeine_funktion (mut irgendeine_person: Person) -> Person {
    println!("Person aus Funktion: {}", irgendeine_person.name );
    irgendeine_person.konto = Konto { guthaben: 999_888 };
    irgendeine_person.name = String::from("neuer Name");
    irgendeine_person
}

fn main() {

    let konto1 = Konto { guthaben: 123_321_444 };
    let person1 = Person {name: String::from("Tibor"), konto: konto1 };

    println!("Person: {}", person1.name );
    println!("Konto von: {} hat Guthaben mit: {} ",person1.name, person1.konto. guthaben);

    let neue_person = irgendeine_funktion(person1);

    println!("Neue Person: {}", neue_person.name );
    println!("Konto von: {} hat Guthaben mit: {} ",neue_person.name, neue_person.konto. guthaben);

    //println!("Alte Person: {}", person1.name );
    //println!("Altes Konto von: {} hat Guthaben mit: {} ",person1.name, person1.konto. guthaben);
}
```

Ausgabe:

```
Person: Tibor
Konto von: Tibor hat Guthaben mit: 123321444
Person aus Funktion: Tibor
Neue Person: neuer Name
Konto von: neuer Name hat Guthaben mit: 999888
```

Versuchen Sie die beiden untersten auskommentierten Zeilen auszukommentieren und das Programm zu kompilieren. Sie werden die Fehlermeldung „value borrowed after move erhalten“. Sie können nicht, wie beispielsweise in Java, dann immer noch auf die alten Referenzen zugreifen. Hier wurde die sogenannte „Ownership“ verschoben („moved“).

102 Beispiel Struct mit Verwendung in einer Funktion:

```
struct Konto {  
    guthaben: i64,  
}  
  
fn irgendeine_funktion (konto: Konto) {  
    println!("Konto in Funktion: {}", konto.guthaben );  
}  
  
fn main() {  
    let konto1 = Konto { guthaben: 123_321_444 };  
    let konto_borrowed = &konto1;  
  
    println!("Konto1: {}", konto1.guthaben);  
    println!("Konto_borrowed: {}", konto_borrowed.guthaben);  
  
    irgendeine_funktion(konto1); // kann über Methode aufgerufen werden  
    //irgendeine_funktion(konto_borrowed); // kann nicht aufgerufen werden  
}
```

Ausgabe:

```
Konto1: 123321444  
Konto_borrowed: 123321444  
Konto in Funktion: 123321444
```

konto_borrowed ist eben nur ausgeliehen und kann deswegen nicht der Funktion übergeben werden. Probieren Sie aus welche Meldung Sie bekommen, wenn sie die unteren beiden Zeilen ein- bzw. auskommentieren. Die Fehlermeldung ist: „expected struct `Konto`, found `&Konto`“ .

103 Beispiel Struct mit Verwendung in einer Funktion und borrowed:

```
struct Konto {  
    guthaben: i64,  
}  
  
fn irgendeine_funktion (konto: Konto) {  
    // konto.guthaben = 0; // geht nur wenn als Parameter "mut konto: Konto" anegeben wird  
    println!("Konto in Funktion: {}", konto.guthaben );  
}  
  
fn main() {  
    let mut konto1 = Konto { guthaben: 123_321_444 };  
    let konto_borrowed = &mut konto1;  
  
    // konto1.guthaben = 333; // geht nicht weil borrowed  
    konto_borrowed.guthaben = 522;  
    //println!("Konto1: {}", konto1.guthaben); // kann hier nicht aufgerufen werden  
    println!("Konto_borrowed: {}", konto_borrowed.guthaben);  
  
    irgendeine_funktion(konto1); // kann über Methode aufgerufen werden  
    //irgendeine_funktion(konto_borrowed); // kann nicht aufgerufen werden  
}
```

Ausgabe:

```
Konto_borrowed: 522  
Konto in Funktion: 522
```

Probieren Sie sämtliche Konstellationen aus.

104 Beispiel FEHLER!!! Struct borrowed:

Folgendes lässt sich nicht kompilieren. Probieren Sie mal alle Konstellationen aus, wenn Sie das Initialisieren von `konto1` und `konto_borrowed` so lassen wie es ist.

```
// Achtung diese Programm laesst sich so nicht kompilieren!!!
struct Konto {
    guthaben: i64,
}

fn main() {
    let mut konto1 = Konto { guthaben: 123_321_444 };
    let konto_borrowed = &mut konto1;

    konto1.guthaben = 333;
    konto_borrowed.guthaben = 522;
    println!("Konto1: {}", konto1.guthaben);
    println!("Konto_borrowed: {}", konto_borrowed.guthaben);
}
```

Ausgabe:

```
error[E0506]: cannot assign to `konto1.guthaben` because it is borrowed
--> src\main.rs:9:5
|
7 |     let konto_borrowed = &mut konto1;
|                           ----- borrow of `konto1.guthaben` occurs here
8 |
9 |     konto1.guthaben = 333;
|     ^^^^^^^^^^^^^^^^^^^^^^^ assignment to borrowed `konto1.guthaben` occurs here
10 |     konto_borrowed.guthaben = 522;
|     ----- borrow later used here
```

Probieren Sie alle Konstellationen aus, indem Sie die unteren 4 Zeilen mal einkommentieren und auskommentieren.

105 Beispiel enum :

```
#[derive(Debug)]

enum Farbe {
    Gruen,
    Rot,
    Blau,
    Gelb,
}

fn main() {
    let meine_farbe = Farbe::Blau;
    println!("meineFarbe: {:?}", meine_farbe);
}
```

Ausgabe:

```
meineFarbe: Blau
```

In einem enum können Sie immer genau ein Element aus dem Enum setzen. Nehmen Sie diesmal `{:?}` um das enum ausgeben zu können `#[derive(Debug)]` dazu.

106 Beispiel: expect, Datei einlesen (Windows):

```
use std::fs::File;
use std::fs;

fn main(){

    let mut data = String::new();
    let text_in_file = fs::read_to_string(r"C:\\resources\\myFile.txt")
        .expect("Datei habe ich nicht gefunden");

    println!("Text gelesen: {} ", text_in_file);
}
```

Datei (in Windows unter C:\\resources\\myFile.txt):

Mein Text in meinem File.

Ausgabe:

Text gelesen: Mein Text in meinem File.

Probieren Sie aus, was passiert, wenn der Dateipfad nicht existiert.

107 Beispiel: Datei einlesen (Linux):

```
use std::fs;
use std::io::Read;

fn main() {
    let result_as_bytes = fs::read("./src/resources/myfile.txt").expect("reading file error");
    let text_from_bytes = String::from_utf8(result_as_bytes).expect("error");
    println!("Text: {}", text_from_bytes);
}
```

Datei (in Windows unter `C:\\resources\\myFile.txt`):

```
[package]
name = "line_test"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

Ausgabe:

```
Text: Text1
```

Probieren Sie aus, was passiert, wenn der Dateipfad oder die Datei nicht existiert.

108 Beispiel Result :

```
fn check_result(my_value:i8) -> Result<i8,String> {
  if my_value == 2 {
    Ok(100)
  } else {
    Err(String::from("der Übergebene Wert entspricht keinem i8"))
  }
}

fn main() {
  let checked_result = check_result(2);

  match checked_result {
    Ok(n) => println!("Wert ist {}", n),
    Err(err) => println!("Error ist eingetreten: {}",err),
  }

  println!();

  let checked_result = check_result(-22);

  match checked_result {
    Ok(n) => println!("Wert ist {}", n),
    Err(err) => println!("Error ist eingetreten: {}",err),
  }
}
```

Ausgabe:

Wert ist 100

Error ist eingetreten: der Übergebene Wert entspricht keinem i8

Ein Result ist ein Enum, das , wenn es zurückgegeben wird, entweder ein OK mit Wert, oder ein Error (Err) zurückliefert. Auch die mein-Funktion kann ein Result zurückgeben.

109 Beispiel Result mit ? (Elvis-Operator, ähnlich dem vorherigen Beispiel) :

```
fn check_result(my_value:i8) -> Result<i8,String> {
    if my_value == 2 {
        Ok(100)
    } else {
        Err(String::from("der Übergebene Wert entspricht keinem i8"))
    }
}

fn main() -> Result<(),String> {

    let r = check_result(2)?;
    //let r = check_result(-22)?;
    println!("r: {}", r);
    Ok(())
}
```

Ausgabe bei mit 2 als Argument:

```
r: 100
```

Ausgabe bei mit -22 als Argument:

```
Error: "der Übergebene Wert entspricht keinem i8"
error: process didn't exit successfully: `target\debug\SimpleListAndForLooop.exe` (exit code: 1)
```

Probieren Sie sowohl die -22 und die 2 aus, bzw. irgendwelche Zahlen die zwischen -128 und 127 liegen. Probieren Sie auch was passiert, wenn Sie einen String übergeben wollen.

110 Beispiel Result mit unwrap (ähnlich dem vorherigen Beispiel) :

```
fn check_result(my_value:i8) -> Result<i8,String> {
    if my_value == 2 {
        Ok(100)
    } else {
        Err(String::from("der Übergebene Wert entspricht keinem i8"))
    }
}

fn main() -> Result<(),String> {

    // Probieren Sie auch die -22
    let r = check_result(2).unwrap();
    println!("r unwrapped {}", r);

    Ok(())
}
```

Ausgabe bei mit 2 als Argument:

```
r unwrapped 100
```

Ausgabe bei mit -22 als Argument:

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: "der Übergebene Wert entspricht keinem i8"',
src\main.rs:12:31
stack backtrace:
 0: std::panicking::begin_panic_handler
   at /rustc/7737e0b5c4103216d6fd8cf941b7ab9bdbaace7c\library\src\panicking.rs:584
 1: core::panicking::panic_fmt
   at /rustc/7737e0b5c4103216d6fd8cf941b7ab9bdbaace7c\library\core\src\panicking.rs:143
 2: core::result::unwrap_failed
   at /rustc/7737e0b5c4103216d6fd8cf941b7ab9bdbaace7c\library\core\src\result.rs:1749
 3: enum$<core::result::Result<i8,alloc::string::String> >::unwrap<i8,alloc::string::String>
   at /rustc/7737e0b5c4103216d6fd8cf941b7ab9bdbaace7c\library\core\src\result.rs:1065
 4: a001SimpleListAndForLooop::main
   at .\src\main.rs:12
 5: core::ops::function::FnOnce::call_once<enum$<core::result::Result<tuple$<>,alloc::string::String>, 1,
18446744073709551615, Err> (*)(),tuple$<> >
   at /rustc/7737e0b5c4103216d6fd8cf941b7ab9bdbaace7c\library\core\src\ops\function.rs:227
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
error: process didn't exit successfully: `target\debug\a001SimpleListAndForLooop.exe` (exit code: 101)

Process finished with exit code 101
```

Mit unwrap erhalten Sie einen Wert zurück im OK Fall, bzw. Panic im Nicht-OK Fall. Panic bedeutet, dass das Programm komplett beendet wird.

111 Beispiel enum :

```
use std::fmt;
use std::fmt::Debug;
use std::any::Any;
use std::borrow::BorrowMut;

enum Konto_Typ {
    Privat,
    Geschaefthlich,
    Sparkonto,
}

impl fmt::Debug for Konto_Typ {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            Konto_Typ::Sparkonto => write!(f, "Sparkonto"),
            Konto_Typ::Privat => write!(f, "Privatkonto"),
            Konto_Typ::Geschaefthlich => write!(f, "Geschaefthlich"),
        }
    }
}

struct Konto {
    eigentuemer: String,
    guthaben: i64, // hier gibt es ein Warning, warum?
    konto_typ: Konto_Typ // hier gibt es ein Warning, warum?
}

fn main() {
    let mein_konto = Konto {
        eigentuemer: String::from("T.Kalman"),
        guthaben: -8200,
        konto_typ: Konto_Typ::Privat
    };

    let mut a = 0;
    match mein_konto.konto_typ {
        Konto_Typ::Geschaefthlich => a = 1,
        Konto_Typ::Privat => a = 2,
        Konto_Typ::Sparkonto => a = 3,
    }

    println!("a: {} ", a);

    if a == 2 {
        println!("Kontotyp: {:#?}, guthaben: {}", Konto_Typ::Privat, mein_konto.guthaben);
    } else {
        println!("Kontotyp ist nicht privat guthaben: {}", mein_konto.guthaben);
    }
    println!("Eigentümer: {}", mein_konto.eigentuemer);
}
```

Ausgabe:

a: 2

Kontotyp: Privatkonto, guthaben: -8200

Eigentümer: T.Kalman

Es gibt mehrere Warnings, weil z.B. kein Sparkonto angelegt wird. Es ist notwendig Debug zu implementieren, da sonst der Konto_Typ nicht ausgegeben werden kann. Spielen Sie etwas mit diesem Programm herum.

112 Beispiel match :

```
fn main() {  
  let meine_farbe:i32 = 5;  
  
  let mein_match = match meine_farbe {  
    1 => "Zahl Eins",  
    2 => "Zahl Zwei",  
    _ => "_ steht für default"  
  };  
  println!("meineFarbe: {}", meine_farbe);  
}
```

Ausgabe:

```
meineFarbe: 5
```

Das Programm gibt nicht das Richtige aus. Bitte korrigieren Sie das Programm so, dass es mein_match ausgibt
Match funktioniert ähnlich wie switch in C++ und Java.

113 Beispiel match mit oder:

```
fn main() {  
    let meine_farbe:i32 = 5;  
  
    let mein_match = match meine_farbe {  
        1 => "Zahl Eins",  
        2 => "Zahl Zwei",  
        3 | 5 => "Zahl ist drei oder sie ist fünf",  
        _ => "_ steht für default"  
    };  
    println!("mein_match: {}", mein_match);  
}
```

Ausgabe:

```
mein_match: Zahl ist drei oder sie ist fünf
```

Mit dem | „oder“ Zeichen kann ich sagen, das ein Wert entweder oder haben soll, bevor ich mein_match etwas zuweise.

114 Beispiel match :

```
fn main() {  
    let meine_farbe:i32 = 5;  
  
    let mein_match = match meine_farbe {  
        1 => "Zahl Eins",  
        2 => "Zahl Zwei",  
        3 ... 5 => "Zahl liegt zwischen drei und fünf",  
        _ => "_ steht für default"  
    };  
    println!("mein_match: {}", mein_match);  
}
```

Ausgabe:

```
mein_match: Zahl liegt zwischen drei und fünf
```

Mit beispielsweise `3 ... 5`
kann man angeben, dass eine Zahl zwischen 3 und 5 liegen soll.
Das ganze geht auch mit Charactern also z.B. `'r' ... 't'`.

115 Beispiel match mit einer bounded variable:

```
fn main() {  
    let a = 50;  
  
    match a {  
        50 => {  
            println!("zahl");  
        }  
        my_bounded_variable @ 3..=53 => {  
            println!("my_bounded_variable: {}", my_bounded_variable);  
        }  
        _ => {  
            println!("default");  
        }  
    }  
}
```

Ausgabe:

```
zahl
```

Achten Sie darauf, dass sowohl die erste Bedingung und die zweite Bedingung zutreffen, jedoch nur die erste ausgeführt wird.

Deswegen werden wir jetzt die Bedingungen umdrehen um zu sehen, dass die Reihenfolge eine Rolle spielt und wir durchaus darauf achten müssen:

```
fn main() {  
    let a = 50;  
  
    match a {  
        my_bounded_variable @ 3..=53 => {  
            println!("my_bounded_variable: {}", my_bounded_variable);  
        }  
        50 => {  
            println!("zahl");  
        }  
        _ => {  
            println!("default");  
        }  
    }  
}
```

Ausgabe:

```
my_bounded_variable: 50
```

116 Beispiel struct und impl :

```
struct Summe {  
    a:f32,  
    b:f32,  
}  
  
impl Summe {  
    fn berechne (&self) -> f32 {  
        self.a + self.b  
    }  
}  
  
fn main() {  
    let sum = Summe{a:20.4, b:30.2};  
    println!("sum: {}", sum.berechne());  
}
```

Ausgabe:

```
sum: 50.6
```

Mit impl können innerhalb eines structs Funktionen angegeben werden. Diese nennen wir in diesem Zusammenhang dann Methode. Es dürfen mehrere verschiedene impl-Blöcke vorhanden sein, auch wenn es nicht notwendig ist.

117 Beispiel Konstruktor (assoziierte Funktion):

```
struct Summe {  
    a:f32,  
    b:f32,  
}  
  
impl Summe {  
    fn new () -> Summe {  
        Summe{a:20.4, b:30.2}  
    }  
  
    fn berechne (&self) -> f32 {  
        self.a + self.b  
    }  
}  
  
fn main() {  
    let sum = Summe::new();  
    println!("sum: {}", sum.berechne());  
}
```

Ausgabe:

```
sum: 50.6
```

Hier ist ein Beispiel für das Programm mittels Konstruktor (new). Der ::-Operator deutet zum Aufruf einer statischen Methode. Der Konstruktor-Aufruf eines Structs ist also ein statischer Aufruf. Die Methode berechne, wird dann mit dem Punkt-Operator aufgerufen, nachdem eine Instanz erzeugt wurde (mit new. Hier heisst die Instanz „sum“). Man nennt dies auch assoziierte Funktion (sie muss nicht „new“ heissen, sondern kann irgendeinen Namen erhalten). Es können auch mehrere solcher Funktionen definiert werden.

118 Beispiel Vector :

```
fn main() {  
    let mein_vector = vec![8,4,3];  
    println!("mein_vector: {}", mein_vector[2]);  
}
```

Ausgabe:

```
mein_vector: 3
```

Ein Vector in Rust ist ähnlich wie ein Vector in Java. Rust ist allerdings nicht in der Lage zur Compilezeit einen Zugriff auf eine Stelle die im Vector nicht existiert zu verhindern. Die Integer Zahlen die man hier sieht sind allerdings vom Typ `usize`.

119 Beispiel Vector, Elemente mit for-Schleife darstellen:

```
fn main() {  
    let mein_vec = vec![1, 2, 3];  
    for e in &mein_vec {  
        println!("Wert: {}", e);  
    }  
}
```

Ausgabe:

```
Wert: 1  
Wert: 2  
Wert: 3
```

Sie sehen hier einfach eine simple For-Schleife, die die einzelnen Elemente darstellt.

120 Beispiel Vector, Elemente mit for-Schleife verändern und darstellen:

```
fn main() {  
    let mut mein_vec = vec![1, 2, 3];  
    for e in &mut mein_vec {  
        *e = *e + 2; // oder *e += 2;  
        println!("Wert: {}", e);  
    }  
}
```

Ausgabe:

```
Wert: 3  
Wert: 4  
Wert: 5
```

Um in einer For-Schleife, Elemente zu ändern, bedarf es dem Keyword `mut` und dem `*`-Operator. Er wird Dereferenzierungsoperator genannt.

121 Beispiel Vector :

```
fn main() {  
    let mein_vector : Vec<u64> = vec![10, 21, 42, 3, 141, 15];  
  
    let element_an_stelle_1: Option<&u64> = mein_vector.get(1);  
  
    match element_an_stelle_1 {  
        Some(element_an_stelle_1) => println!("Wert an Stelle 1: {}", element_an_stelle_1),  
        None => println!("Dieses None muss hier stehen. Probieren Sie aus, was passiert wenn es fehlt."),  
    }  
}
```

Ausgabe:

```
Wert an Stelle 1: 21
```

Versuchen Sie an einer der beiden Stellen, an denen u64 angegeben wird i64 oder i32 oder u32 zu schreiben.

122 Beispiel Vector :

```
fn main() {  
    let mein_vector : Vec<u64> = vec![10, 21, 42, 3, 141, 15];  
  
    let element_an_stelle_4: &u64 = &mein_vector[4];  
  
    println!("Wert an Stelle 4: {}", element_an_stelle_4);  
}
```

Ausgabe:

```
Wert an Stelle 4: 141
```

Hier haben Sie eine andere Möglichkeit gesehen, wie man an den Wert in einem Vector an einer bestimmten Stelle drankommt.

123 Beispiel Vector Element hinzufügen :

```
fn main() {  
    let mut mein_vector: Vec<usize>= vec![8,4,3];  
    mein_vector.push(44);  
    println!("mein_vector: {}", mein_vector[3]);  
}
```

Ausgabe:

```
mein_vector: 44
```

Will man dynamisch ein Element hinzufügen, dann kann man dies bei einem Vector mit push tun.

Aufgabe:

Probieren Sie mal den Befehl `vector.reverse()` aus. Er dreht einfach die Ausgabe um, also von hinten nach vorne. Der Vector Inhalt wird aber nicht umgedreht, nur die Ausgabe.

124 Beispiel Vector Element löschen :

```
fn main() {  
    let mut mein_vector: Vec<usize>= vec![8,4,3];  
    let t = mein_vector.pop();  
    println!("geloeschte Element: {:?}", t);  
}
```

Ausgabe:

```
geloeschte Element: Some(3)
```

Beim Löschen aus einem Vector wird ein Some zurückgegeben. Dies können wir mittels {:?} mit println ausgeben.

125 Beispiel Vector mit Macro anlegen und Iterator ausgeben :

```
fn main() {  
    let mut mein_vector:Vec<i8>= vec![8,4,3];  
    mein_vector.push(44);  
  
    for my_value_from_iterator in mein_vector.iter() {  
        println!("{}", my_value_from_iterator);  
    }  
}
```

Ausgabe:

```
8  
4  
3  
44
```

Beim Anlegen eines Vectors kann man über einen Iterator die einzelnen Werte hindurchgehen.

126 Beispiel Vector, Iterator nach bestimmten Feldern sammeln :

```
#[derive(Debug)]
struct Konto {
    guthaben: f64,
    kontonummer: i32
}

fn kontos_by_guthaben_collect( gesuchtes_guthaben: f64, kontos: Vec<Konto>) -> Vec<Konto> {
    kontos.into_iter()
        .filter(|guth| guth.guthaben != gesuchtes_guthaben)
        .collect()
}

fn main() {
    let mut konto_liste = vec![
        Konto { guthaben: 1011.67, kontonummer: 345 },
        Konto { guthaben: 234.0, kontonummer: 123 },
        Konto { guthaben: -234.9, kontonummer: 1 },
        Konto { guthaben: 33.0, kontonummer: 9 },
        Konto { guthaben: 1011.67, kontonummer: 897 },
    ];

    let kontos_mit_guthaben = kontos_by_guthaben_collect( 1011.67, konto_liste);

    for k in kontos_mit_guthaben.iter() {
        println!("{:#?} Konto {}:", k, k.kontonummer);
    }
}
```

Ausgabe:

```
Konto {
  guthaben: 234.0,
  kontonummer: 123,
} Konto 123:
Konto {
  guthaben: -234.9,
  kontonummer: 1,
} Konto 1:
Konto {
  guthaben: 33.0,
  kontonummer: 9,
} Konto 9:
```

In diesem Beispiel sehen wir, wie wir aus einem Vector von mehreren Objekten z.B. nach einzelnen Elementen suchen können die nicht einer Bedingung entsprechen. Also alle Konto, die kein Guthaben von „x“ haben.

Beispiel abgeleitet aus:

<https://rust-lang-de.github.io/rustbook-de/ch13-02-iterators.html>

127 Beispiel Vector mit new anlegen und Iterator ausgeben :

```
fn main() {  
    let mut mein_vector:Vec<i8>= Vec::new();  
    // oder let mut mein_vector= Vec::new();  
    // oder let mut mein_vector = Vec::i8>::new();  
    // oder let mut mein_vector:Vec<i8> = Vec::i8>::new();  
  
    mein_vector.push(44);  
    mein_vector.push(11);  
  
    for my_value_from_iterator in mein_vector.iter() {  
        println!("{}", my_value_from_iterator);  
    }  
}
```

Ausgabe:

```
44  
11
```

Der Vector wird hier mit new angelegt und mittels Iterator in einer for-Schleife weider ausgegeben.

128 Beispiel HashMap :

```
use std::collections::HashMap;

fn main() {

    let mut meine_hash_map = HashMap::new();

    meine_hash_map.insert(String::from("Erster Eintrag"), 22);
    meine_hash_map.insert(String::from("Zweiter"), 11);
    meine_hash_map.insert(String::from("Dritter"), 33);

    let zweiter_eintrag = String::from("Zweiter");

    let wert = meine_hash_map.get(&zweiter_eintrag);
    println!("Wert: {}", wert.unwrap());

    let dritter_eintrag = String::from("Dritter");
    let wert = meine_hash_map.get(&dritter_eintrag);
    println!("Wert: {}", wert.unwrap());
}
```

Ausgabe:

```
Wert: 11
Wert: 33
```

Die HashMap ermöglicht ähnlich wie bei einem Vector mehrerer Elemente in einer Variablen abzulegen. Allerdings kann /muss man noch einen key angeben, dies kann wie hier ein String sein, kann aber auch andere Typen haben.

129 Beispiel HashMap, Ausgabe über for-Schleife :

```
use std::collections::HashMap;

fn main() {

    let mut meine_hash_map = HashMap::new();

    meine_hash_map.insert(String::from("Erster Eintrag"), 22);
    meine_hash_map.insert(String::from("Zweiter"), 11);
    meine_hash_map.insert(String::from("Dritter"), 33);

    for (schluessel, wert) in &meine_hash_map {
        println!("Schlüssel: {}", schluessel);
        println!("Wert:{}", wert);
        println!("{}", "");
    }
}
```

Ausgabe:

```
Schlüssel: Dritter
Wert:33

Schlüssel: Erster Eintrag
Wert:22

Schlüssel: Zweiter
Wert:11
```

Hier sieht man, wie man über eine HashMap iteriert und dabei Wert und Schlüssel herausholen kann.

130 Beispiel HashMap, insert eines vorhandenen key :

```
use std::collections::HashMap;

fn main() {

    let mut meine_hash_map = HashMap::new();

    meine_hash_map.insert(String::from("Erster Eintrag"), 22);
    meine_hash_map.insert(String::from("Zweiter"), 11);
    meine_hash_map.insert(String::from("Erster Eintrag"), 33);

    for (schluessel, wert) in &meine_hash_map {
        println!("Schlüssel: {}", schluessel);
        println!("Wert: {}", wert);
        println!("{}", "");
    }
}
```

Ausgabe:

```
Schlüssel: Erster Eintrag
Wert:33

Schlüssel: Zweiter
Wert:11
```

Ein Eintrag der schon vorhanden ist wird mit insert einfach überschrieben. Er ist dann auch nur einmal in der HashMap.

131 Beispiel Shadowing : (2022-06-19)

```
fn main() {  
  let m_v:u8 = 255;  
  println!("m_v: {} ", m_v );  
  
  let m_v:u16 = 100;  
  println!("m_v: {} ", m_v );  
}
```

Ausgabe:

```
m_v: 255  
m_v: 100
```

Sobald eine Variable schon einmal verwendet wurde, kann sie unter genau denselben Namen wiederverwendet werden, dabei spielt der Typ keine Rolle, auch nicht wenn die Variable veränderbar gemacht wurde (z.B. mit `mut`). Es gibt allerdings dann Warnings, die darauf hinweisen, dass `mut` nicht notwendig ist).

132 Beispiel Shadowing: (2022-07-22)

```
fn main() {  
  let m_v:u8 = 255;  
  println!("m_v: {} ", m_v);  
  {  
    let m_v:u16 = 100;  
    println!("m_v im Block: {} ", m_v);  
  }  
  println!("m_v ausserhalb des Blocks: {} ", m_v);  
}
```

Ausgabe:

```
m_v: 255  
m_v im Block: 100  
m_v ausserhalb des Blocks: 255
```

Auch dies ist ein Beispiel für Shadowing. Beachten Sie, dass dabei nach dem inneren Block die Variable wieder den alten Wert hat.

133 Beispiel Einführung für Generic :

```
fn third_element_int_array(my_array: &[i64]) -> i64 {
    my_array[3]
}

fn third_element_character_array(my_array: &[char]) -> char {
    my_array[3]
}

fn main() {
    let integer_vector = vec![1, 2, 4, -2, 8];
    let third_element = third_element_int_array(&integer_vector);
    println!("Dritte Integer ist: {}", third_element);

    let character_vector = vec!['Q', 'b', 'e', 'r', 't'];
    let third_element = third_element_character_array(&character_vector);
    println!("Dritte Buchstabe ist: {}", third_element);
}
```

Ausgabe:

```
Dritte Integer ist: -2
Dritte Buchstabe ist: r
```

Zwei Funktionen die jeweils das dritte Element aus einem Vector holen, sollen implementiert werden. Da wir DRY verwenden wollen, weil wir innerhalb eines Services sind, sollten wir eigentlich, so wie hier, eine doppelte Implementierung von ein und derselben Art und Weise, vermeiden. Hier wird dieselbe Funktion 2 mal geschrieben, nur weil wir es einmal mit Integern und einmal mit Charactern zu tun haben.

134 Beispiel Generic Type : (2022-06-21)

```
struct MeinStructVonIrgendwas<T> {  
    besitze_ich: T,  
    bezeichnung: String,  
}  
  
fn main() {  
  
    let reichum = MeinStructVonIrgendwas::<String> {  
        besitze_ich: String::from("ich weiss nicht recht, ob ich das will"),  
        bezeichnung: String::from("Geld")  
    };  
    println!("{}", reichum.bezeichnung, reichum.besitze_ich);  
  
    let jacht = MeinStructVonIrgendwas::<bool> { besitze_ich: false, bezeichnung: String::from("Jacht") };  
    println!("{}", jacht.bezeichnung, jacht.besitze_ich);  
  
    let buecher = MeinStructVonIrgendwas::<i8> { besitze_ich: 2, bezeichnung: String::from("Bücher") };  
    println!("{}", buecher.bezeichnung, buecher.besitze_ich);  
}
```

Ausgabe:

```
Geld: ich weiss nicht recht, ob ich das will  
Jacht: false  
Bücher: 2
```

Mit Generic Types lassen sich z.B. aus ein und demselben struct mehrere Instanzen bilden die dann allerdings für bestimmte Felder unterschiedliche Typen haben .

135 Beispiel Option<T> : (2022-06-21)

```
// folgender Programmcode ist dem Beispiel us
// https://tourofrust.com/35_de.html
// abgeleitet

struct Irgendetwas<T> {
    mein_wert: Option<T>,
}

fn main() {

    let irgendeini8 = Irgendetwas::<i8> { mein_wert: None };

    match irgendeini8.mein_wert {
        Some(s) => println!(".. not None.. {}", s),
        None => println!("..None.."),
    }

    let irgendeini8 = Irgendetwas::<i8> { mein_wert: Some(55) };

    match irgendeini8.mein_wert {
        Some(s) => {
            println!(".. not None.. {}", s);
            println!("{}", s)
        },
        None => println!("..None.."),
    }

    let irgendeinf64 = Irgendetwas::<f64> { mein_wert: Some(33.3f64) };

    if irgendeinf64.mein_wert.is_none() {
        println!("is none")
    } else {
        println!("is not none")
    }

    let irgendein_anderes_f64 = Irgendetwas::<f64> { mein_wert: None };

    if irgendein_anderes_f64.mein_wert.is_none() {
        println!("is none")
    } else {
        println!("is not none")
    }
}
```

Ausgabe:

```
..None..
.. not None.. 55

is not none
is none
```

Um so genannte „null“ Werte, die wir beispielsweise nach der Abfrage aus Datenbanken erhalten können (null steht für so viel wie „nichts enthalten“, oder „nicht gesetzt“), gibt es das Enum Option<T>. Statt T können wir allerdings auch einen konkreten Typ setzen. In Rust wollte man null vermeiden, weil in Sprachen wie C, C++ oder Java dies leider sehr

oft zu Programmabstürzen geführt hat, die man nicht wollte, bzw. auch dazu, dass mit nicht gesetzten Werten weitergearbeitet wurde, obwohl man nicht hätte sollen.

Objekte:

Rust ist eine Programmiersprache die nicht objektorientiert ist. D.h. jedoch nicht, dass wir nicht mit Objekten arbeiten könnten. Zur Objektorientierung zählt, z.B. dass Objekte vererbbar sein und Mehrfachvererbung unterstützen müssen, beides tut Rust jedoch nicht. Weder Funktionen, noch Felder können in Rust durch Vererbung einem anderen Konstrukt oder Objekt gegeben werden.

136 Beispiel &self :

```
struct Konto {  
    eigentuemer: String,  
    bezeichnung: String,  
}  
  
impl Konto {  
    fn get_eigentuemer(&self) -> &str {  
        &self.eigentuemer  
    }  
  
    fn get_bezeichnung(&self) -> &str {  
        &self.bezeichnung  
    }  
}  
  
fn main() {  
    //Erstellung eines neuen Objekts vom Typ Konto  
    let konto = Konto {  
        bezeichnung: String::from("Sparkonto"),  
        eigentuemer: String::from("Ich"),  
    };  
    println!("Kontobezeichnung: {}", konto.get_bezeichnung());  
    println!("Eigentümer: {}", konto.get_eigentuemer());  
}
```

Ausgabe:

```
Kontobezeichnung: Sparkonto  
Eigentümer: Ich
```

In Rust gibt es Objekte die Funktionen und Felder haben können. Mit &self kann man innerhalb der Funktionen auf die Felder zugreifen.

137 Beispiel trait :

```
struct SparKonto {
    eigentuemer: String,
    bezeichnung: String,
}

impl SparKonto {
    fn get_eigentuemer(&self) -> &str {
        &self.eigentuemer
    }

    fn get_bezeichnung(&self) -> &str {
        &self.bezeichnung
    }
}

trait KontoAnzeige {
    fn anzeigen(&self);
}

impl KontoAnzeige for SparKonto {
    fn anzeigen(&self) {
        println!("Eigentümer {}, Bezeichnung {}", &self.get_eigentuemer(), &self.get_bezeichnung());
    }
}

fn main() {
    let spar_konto = SparKonto {
        bezeichnung: String::from("Sparkonto"),
        eigentuemer: String::from("Ich"),
    };
    println!("Kontobezeichnung: {}", spar_konto.get_bezeichnung());
    println!("Eigentümer: {}", spar_konto.get_eigentuemer());

    println!("Eigentümer: {}", spar_konto.eigentuemer);
    println!("Bezeichnung: {}", spar_konto.bezeichnung);

    spar_konto.anzeigen();
}
```

Ausgabe:

```
Kontobezeichnung: Sparkonto
Eigentümer: Ich
Eigentümer: Ich
Eigentümer: Sparkonto
Eigentümer Ich, Bezeichnung Sparkonto
```

Ähnlich einem Interface in anderen Sprachen wie z.B. Java lassen sich durch Traits den einzelnen Implementierungen von Struct die Verwendung durch Funktionen erzwingen.

138 Beispiel trait mit 2 structs:

```
struct SparKonto {
    eigentuemer: String,
    bezeichnung: String,
}

impl SparKonto {
    fn get_eigentuemer(&self) -> &str {
        &self.eigentuemer
    }

    fn get_bezeichnung(&self) -> &str {
        &self.bezeichnung
    }
}

struct GiroKonto {
    gebuehr: f64,
}

impl GiroKonto {
    fn get_gebuehr(&self) -> f64 {
        self.gebuehr
    }
}

trait KontoAnzeige {
    fn anzeigen(&self);
}

impl KontoAnzeige for SparKonto {
    fn anzeigen(&self) {
        println!("Eigentümer {}, Bezeichnung {}", &self.get_eigentuemer(), &self.get_bezeichnung());
    }
}

impl KontoAnzeige for GiroKonto {
    fn anzeigen(&self) {
        println!("Gebühr {}", &self.get_gebuehr());
    }
}

fn main() {
    let spar_konto = SparKonto {
        bezeichnung: String::from("Sparkonto"),
        eigentuemer: String::from("Ich"),
    };
    println!("Kontobezeichnung: {}", spar_konto.get_bezeichnung());
    println!("Eigentümer: {}", spar_konto.get_eigentuemer());

    println!("Eigentümer: {}", spar_konto.eigentuemer);
    println!("Bezeichnung: {}", spar_konto.bezeichnung);

    spar_konto.anzeigen();

    let giro_konto = GiroKonto {
        gebuehr: 12.5,
```

```
};  
  
println("");  
  
println("Giro Konto Gebühr: {} ", giro_konto.get_gebuehr());  
giro_konto.anzeigen();  
}
```

Ausgabe:

```
Kontobezeichnung: Sparkonto  
Eigentümer: Ich  
Eigentümer: Ich  
Bezeichnung: Sparkonto  
Eigentümer Ich, Bezeichnung Sparkonto  
  
Giro Konto Gebühr: 12.5  
Gebühr 12.5
```

In diesem Beispiel sieht man, wie ein Trait ermöglicht bei 2 verschiedenen Structs ein und dieselbe Funktion zu erzwingen.

Aufgabe:

Versuchen Sie dem Trait KontoAnzeige einen neuen Prototyp hinzuzufügen ohne den Implementierungen von GiroKonto und SparKonto diese mitzugeben.

139 Beispiel trait mit einer im voll implementierten Funktion:

```
struct SparKonto {
    eigentuemer: String,
    bezeichnung: String,
}

impl SparKonto {
    fn get_eigentuemer(&self) -> &str {
        &self.eigentuemer
    }

    fn get_bezeichnung(&self) -> &str {
        &self.bezeichnung
    }
}

struct GiroKonto {
    gebuehr: f64,
}

impl GiroKonto {
    fn get_gebuehr(&self) -> f64 {
        self.gebuehr
    }
}

trait KontoAnzeige {
    fn anzeigen(&self);

    fn etwas_anderes_tun(&self) {
        println!("etwas anderes tun");
    }
}

impl KontoAnzeige for SparKonto {
    fn anzeigen(&self) {
        println!("Eigentümer {}, Bezeichnung {}", &self.get_eigentuemer(), &self.get_bezeichnung());
    }
}

impl KontoAnzeige for GiroKonto {
    fn anzeigen(&self) {
        println!("Gebühr {}", &self.get_gebuehr());
    }
}

fn main() {
    let spar_konto = SparKonto {
        bezeichnung: String::from("Sparkonto"),
        eigentuemer: String::from("Ich"),
    };
    println!("Kontobezeichnung: {}", spar_konto.get_bezeichnung());
    println!("Eigentümer: {}", spar_konto.get_eigentuemer());

    println!("Eigentümer: {}", spar_konto.eigentuemer);
    println!("Bezeichnung: {}", spar_konto.bezeichnung);
}
```

```

spar_konto.anzeigen();

let giro_konto = GiroKonto {
  gebuehr: 12.5,
};

println("");

println("Giro Konto Gebühr: {} ", giro_konto.get_gebuehr());
giro_konto.anzeigen();

println("");

giro_konto.etwas_anderes_tun();
spar_konto.etwas_anderes_tun();
}

```

Ausgabe:

```

Kontobezeichnung: Sparkonto
Eigentümer: Ich
Eigentümer: Ich
Bezeichnung: Sparkonto
Eigentümer Ich, Bezeichnung Sparkonto

Giro Konto Gebühr: 12.5
Gebühr 12.5

etwas anderes tun
etwas anderes tun

```

In diesem Beispiel sieht man, wie in einem Trait eine Funktion implementiert ist die dann von den jeweiligen Implementierungen des Traits verwendet werden kann. Dis muss dann aber nicht nachimplementiert werden, sondern kann verwendet werden oder nicht.

140 Beispiel Trait mit mit abgeleitetem Trait:

```
struct SparKonto {
    eigentuemer: String,
    bezeichnung: String,
}

impl SparKonto {
    fn get_eigentuemer(&self) -> &str {
        &self.eigentuemer
    }
    fn get_bezeichnung(&self) -> &str {
        &self.bezeichnung
    }
}

struct GiroKonto {
    gebuehr: f64,
}

impl GiroKonto {
    fn get_gebuehr(&self) -> f64 {
        self.gebuehr
    }
}

trait KontoAnzeige {
    fn anzeigen(&self);
}

trait AbgeleiteteKontoAnzeige: KontoAnzeige {
    fn etwas_anderes_tun(&self) {
        println!("etwas anderes tun");
    }

    fn anzeigen2(&self);
}

impl KontoAnzeige for SparKonto {
    fn anzeigen(&self) {
        println!("Eigentümer {}, Bezeichnung {}", &self.get_eigentuemer(), &self.get_bezeichnung());
    }
}

impl AbgeleiteteKontoAnzeige for SparKonto {
    fn anzeigen2(&self) {
        println!("Eigentümer {}", &self.get_eigentuemer());
    }
}

impl KontoAnzeige for GiroKonto {
    fn anzeigen(&self) {
        println!("Gebühr {}", &self.get_gebuehr());
    }
}
```

```

fn main() {
  let spar_konto = SparKonto {
    bezeichnung: String::from("Sparkonto"),
    eigentuemer: String::from("Ich"),
  };
  println!("Kontobezeichnung: {}", spar_konto.get_bezeichnung());
  println!("Eigentümer: {}", spar_konto.get_eigentuemer());

  println!("Eigentümer: {}", spar_konto.eigentuemer);
  println!("Bezeichnung: {}", spar_konto.bezeichnung);

  spar_konto.anzeigen();

  let giro_konto = GiroKonto {
    gebuehr: 12.5,
  };

  println!("");

  println!("Giro Konto Gebühr: {} ", giro_konto.get_gebuehr());
  giro_konto.anzeigen();

  println!("");

  // geht nicht: giro_konto.etwas_anderes_tun();
  spar_konto.etwas_anderes_tun();
}

```

Ausgabe:

```

Kontobezeichnung: Sparkonto
Eigentümer: Ich
Eigentümer: Ich
Bezeichnung: Sparkonto
Eigentümer Ich, Bezeichnung Sparkonto

Giro Konto Gebühr: 12.5
Gebühr 12.5

etwas anderes tun

```

In diesem Beispiel „erbt“ das Trait AbgeleiteteKontoanzeige vom Typ KontoAnzeige. Da Sparkonto davon abgeleitet ist, kann es die Funktion „anzeigen“ verwenden und muss sogar die Funktion „anzeigen2“ durch das Trait „AbgeleiteteKontoAnzeige“ implementieren .

141 Beispiel Trait mit statischem Dispatch:

```
struct SparKonto {
    eigentuemer: String,
    bezeichnung: String,
}

impl SparKonto {
    fn get_eigentuemer(&self) -> &str {
        &self.eigentuemer
    }
    fn get_bezeichnung(&self) -> &str {
        &self.bezeichnung
    }
}

trait KontoAnzeige {
    fn anzeigen(&self);
}

impl KontoAnzeige for SparKonto {
    fn anzeigen(&self) {
        println!("Eigentümer {}, Bezeichnung {}", &self.get_eigentuemer(), &self.get_bezeichnung());
    }
}

// Hier weiss man welcher Traittyp uebergeben wird
fn statischer_dispatch(spar_konto: &SparKonto) {
    println!("spar_konto.bezeichnung: {}", spar_konto.get_bezeichnung());
}

fn main() {
    let spar_konto = SparKonto {
        bezeichnung: String::from("Sparkonto"),
        eigentuemer: String::from("Ich"),
    };

    statischer_dispatch(&spar_konto);
}
```

Ausgabe:

```
spar_konto.bezeichnung: Sparkonto
```

Ein Beispiel, eines Funktionsaufrufs, wenn der Typ eines Traits genau bekannt ist.

142 Beispiel Trait mit dynamischem Dispatch:

```
struct SparKonto {
    eigentuemer: String,
    bezeichnung: String,
}

impl SparKonto {
    fn get_eigentuemer(&self) -> &str {
        &self.eigentuemer
    }
    fn get_bezeichnung(&self) -> &str {
        &self.bezeichnung
    }
}

trait KontoAnzeige {
    fn anzeigen(&self);
}

impl KontoAnzeige for SparKonto {
    fn anzeigen(&self) {
        println!("Eigentümer {}, Bezeichnung {}", &self.get_eigentuemer(), &self.get_bezeichnung());
    }
}

// Hier weiss man nicht welcher Traittyp uebergeben wird, sondern nur das Trait
fn dynamischer_dispatch(unbekannter_typ_des_traits: &dyn KontoAnzeige) {
    unbekannter_typ_des_traits.anzeigen();
}

fn main() {
    let spar_konto = SparKonto {
        bezeichnung: String::from("Sparkonto"),
        eigentuemer: String::from("Ich"),
    };

    dynamischer_dispatch(&spar_konto);
}
```

Ausgabe:

```
Eigentümer Ich, Bezeichnung Sparkonto
```

Ein Beispiel, eines Funktionsaufrufs, wenn der Typ eines Traits nicht genau bekannt ist, sondern nur das Trait. Laut https://tourofrust.com/81_de.html ist dynamisches Dispatching langsamer.

143 Beispiel Trait mit einer generischen Funktion:

```
struct SparKonto {
    eigentuemer: String,
    bezeichnung: String,
}

impl SparKonto {
    fn get_eigentuemer(&self) -> &str {
        &self.eigentuemer
    }
    fn get_bezeichnung(&self) -> &str {
        &self.bezeichnung
    }
}

trait KontoAnzeige {
    fn anzeigen(&self);
}

impl KontoAnzeige for SparKonto {
    fn anzeigen(&self) {
        println!("Eigentümer {}, Bezeichnung {}", &self.get_eigentuemer(), &self.get_bezeichnung());
    }
}

fn generic_function<T>(unbekannter_typ_des_traits: &T)
    where
        T: KontoAnzeige,{
    unbekannter_typ_des_traits.anzeigen();
}

fn main() {
    let spar_konto = SparKonto {
        bezeichnung: String::from("Sparkonto"),
        eigentuemer: String::from("Ich"),
    };

    generic_function(&spar_konto);
}
```

Ausgabe:

```
Eigentümer Ich, Bezeichnung Sparkonto
```

Wir können als Parameter den Typ T nehmen und dabei sagen, welcher geforderte Trait übergeben werden muss. Probieren Sie nun die Funktion mit T durch die untere zu ersetzen:

```
fn generic_function<T>(unbekannter_typ_des_traits: &T)
    where
        T: KontoAnzeige,{..
```

ist also dasselbe wie

```
fn generic_function(unbekannter_typ_des_traits: &impl KontoAnzeige){...
```

Wie Sie sehen ist es dasselbe.

```
struct SparKonto {
    eigentuemer: String,
    bezeichnung: String,
}

impl SparKonto {
    fn get_eigentuemer(&self) -> &str {
        &self.eigentuemer
    }
    fn get_bezeichnung(&self) -> &str {
        &self.bezeichnung
    }
}

trait KontoAnzeige {
    fn anzeigen(&self);
}

impl KontoAnzeige for SparKonto {
    fn anzeigen(&self) {
        println!("Eigentümer {}, Bezeichnung {}", &self.get_eigentuemer(), &self.get_bezeichnung());
    }
}

fn generic_function(unbekannter_typ_des_traits: &impl KontoAnzeige) {
    unbekannter_typ_des_traits.anzeigen();
}

fn main() {
    let spar_konto = SparKonto {
        bezeichnung: String::from("Sparkonto"),
        eigentuemer: String::from("Ich"),
    };

    generic_function(&spar_konto);
}
```

144 Beispiel Box:

```
struct SparKonto {
    eigentuemer: String,
    bezeichnung: String,
}

impl SparKonto {
    fn get_eigentuemer(&self) -> &str {
        &self.eigentuemer
    }
    fn get_bezeichnung(&self) -> &str {
        &self.bezeichnung
    }
}

trait KontoAnzeige {
    fn anzeigen(&self);
}

impl KontoAnzeige for SparKonto {
    fn anzeigen(&self) {
        println!("Eigentümer {}, Bezeichnung {}", &self.get_eigentuemer(), &self.get_bezeichnung());
    }
}

fn main() {

    //let mut sparkonten : Vec<Box<SparKonto>> = Vec::new();
    //let mut sparkonten : Vec<Box<dyn KontoAnzeige>> = Vec::new();
    let mut sparkonten_vector : Vec<Box<dyn KontoAnzeige>>; // = Vec::new();

    let spar_konto_1 = SparKonto {
        bezeichnung: String::from("Sparkonto1"),
        eigentuemer: String::from("Ich"),
    };

    let spar_konto_2 = SparKonto {
        bezeichnung: String::from("Sparkonto2"),
        eigentuemer: String::from("auch meins"),
    };

    sparkonten_vector = vec![Box::new(spar_konto_1), Box::new(spar_konto_2)];

    for konto in sparkonten_vector.iter() {
        konto.anzeigen();
    }
}
```

Ausgabe:

```
Eigentümer Ich, Bezeichnung Sparkonto1
Eigentümer auch meins, Bezeichnung Sparkonto2
```

Um z.B. ein Objekt nicht im Stack zu halten, sondern auf dem Heap.

D.h. dass eine Box einfach nur auf die Daten eines Objekts im Heap zeigt.

So als ob wir auf einem Blatt Papier aufschreiben, was wir in einem Lager wo finden.

D.h. die Grösse einer Box ist klein und gibt einfach nur den Ort an wo etwas im Heap befindet.

Was sich dann auf dem Heap befindet kann unterschiedlich gross sein. Man kann wenn man z.B. mehrere Objekte anlegt dann diese mit einer Box wieder ansprechen.

Z.B. Legt man mehrere Sparkonten an, die man in einer Collection sammeln kann indem man die Objekte über eine Box der Collection hinzufügt.

145 Beispiel: Hängende Referenz:

```
fn main() {  
    let x;  
    {  
        let y = 33;  
        println!("y: {}", y);  
        x = &y;  
        println!("x: {}", x);  
    }  
    println!("x: {}", x);  
}
```

Ausgabe:

```
error[E0597]: `y` does not live long enough  
--> src/main.rs:505:13  
    |  
505 |     x = &y;  
    |         ^^ borrowed value does not live long enough  
506 |     println!("x: {}", x);  
507 | }  
    | - `y` dropped here while still borrowed  
508 | println!("x: {}", x);  
    |         - borrow later used here
```

Kommentieren Sie die println aus und danach Schritt für Schritt alle 3 nacheinander wieder ein. Erst wenn Sie das letzte println einkommentieren, werden Sie den Fehler bekommen, weil der Inhalt von y dem x ausserhalb der geschweiften Klammern nicht mehr zur Verfügung steht. Die Ebene auf der x definiert wird, nennt man beispielsweise 'a und die Ebene in der y definiert wird 'b. Durch explizite Angabe der Lebensdauer, können wir dem Lebensdauer Prüfer sagen, dass er die Lebensdauer an bestimmten Stellen nicht prüfen soll (wir können Sie nicht ändern, nur prüfen).

146 Beispiel: Closure (einfaches Beispiel)

```
fn main() {  
    //let my_closure = |x| { 33.0 + x * 2 };  
    let my_closure = |x: f32| -> f32 { 33.0 + x * 2f32 };  
    println!("result: {}", my_closure(1.5));  
}
```

Ausgabe:

```
result: 36
```

Probieren Sie aus, was passiert, wenn Sie den oberen momentan auskommentierten Code weidereinkommentieren und dafür die untere Definition auskommentieren. Wenn kein Typ mit angegeben wird, wird in diesem Fall ein Integer erwartet.

147 Beispiel: Closure (als Argument)

```
use std::fmt::Display;

fn closures_as_arguments<F: Fn(f32) -> f32>(a: f32, my_function: F) -> f32 {
    my_function(a) + my_function(a)
}

fn main() {
    let my_closure = |b: f32| { b + b };

    let result = closures_as_arguments(2.3, my_closure);

    println!("result: {}", result);
}
```

Ausgabe:

```
result: 9.2
```

Aus meiner Sicht machen solche Konstrukte einen Sourcecode nicht unbedingt übersichtlich, allerdings gibt es Ähnliches inzwischen auch in Java. Dort gibt es Functional Interfaces. Beides ist nicht dasselbe, wird teilweise allerdings synonym verwendet. Closures und Lambda-Funktionen, sind sich ähnlich, Closures können aber Variablen nutzen, die aus dem aktuellen Scope erreichbar sind.

148 Beispiel: Closure (als Argument)

```
#[derive(Debug)]
struct Konto {
    guthaben: f64,
    kontonummer: i32
}

fn main() {

    let mut konto_liste = [
        Konto { guthaben: 1000.0, kontonummer: 345 },
        Konto { guthaben: 234.0, kontonummer: 123 },
        Konto { guthaben: -234.9, kontonummer: 1 },
        Konto { guthaben: 33.0, kontonummer: 9 }
    ];

    let mut guthaben_summe = 0.0;
    konto_liste.sort_by_key(|k| {
        guthaben_summe += k.guthaben;
        k.kontonummer
    });

    println!("Summe aller Guthaben: {}", guthaben_summe);

    for k in &konto_liste {
        println!("{}", k.sortiert_nach_Kontonummer, k);
    }
}
```

Ausgabe:

```
Summe aller Guthaben: 3096.3
Konto {
    guthaben: -234.9,
    kontonummer: 1,
} sortiert nach Kontonummer
Konto {
    guthaben: 33.0,
    kontonummer: 9,
} sortiert nach Kontonummer
Konto {
    guthaben: 234.0,
    kontonummer: 123,
} sortiert nach Kontonummer
Konto {
    guthaben: 1000.0,
    kontonummer: 345,
} sortiert nach Kontonummer
```

Wie man mithilfe eines Closures eine Liste nach einem Element sortieren kann und gleichzeitig z.B. etwas mitzählen kann können Sie im oberen Beispiel sehr gut sehen.

149 Beispiel: Threads

```
use std::time::Duration;
use std::thread;

fn main() {
    thread::spawn(|| {
        for x in 0..50 {
            println!("Thread 1: {}", x);
            thread::sleep(Duration::from_millis(6));
        }
    });

    thread::spawn(|| {
        for y in 0..50 {
            println!("Thread 2: {}", y);
            thread::sleep(Duration::from_millis(50));
        }
    });

    let mut ii:i32 = 0;
    loop {
        println!("Thread loop Main: {}", ii);
        thread::sleep(Duration::from_millis(20));
        ii+=1;
        if ii > 49 { break; }
    }

    thread::spawn(|| {
        for j in 0..50 {
            println!("Thread 3: {}", j);
            thread::sleep(Duration::from_millis(1));
        }
    });
}
```

Ausgabe:

```
Thread loop Main: 0
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 1: 2
Thread loop Main: 1
Thread loop Main: 2
Thread 2: 1
Thread loop Main: 3
Thread 2: 2
Thread loop Main: 4
```

Achten Sie darauf, dass Ihr Ergebnis nach jedem Aufruf anders aussehen kann. In unserem Beispiel kann es vorkommen, dass Thread 3 zum Zug kommt, oder auch nicht.

150 Beispiel: Threads

```
use std::time::Duration;
use std::thread;

fn main() {
    thread::spawn(|| {
        for x in 0..3 {
            println!("Thread 1: {}", x);
            thread::sleep(Duration::from_millis(6));
        }
    });

    let my_join_handle = thread::spawn(|| {
        for y in 0..3 {
            println!("Thread 2: {}", y);
            thread::sleep(Duration::from_millis(50));
        }
    });

    let mut ii:i32 = 0;
    loop {
        println!("Thread loop Main: {}", ii);
        thread::sleep(Duration::from_millis(20));
        ii+=1;
        if ii > 4 { break; }
    }

    thread::spawn(|| {
        for j in 0..33 {
            println!("Thread 3: {}", j);
            thread::sleep(Duration::from_millis(1));
        }
    });

    my_join_handle.join().unwrap();
}
```

Ausgabe:

```
Thread loop Main: 0
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread loop Main: 1
Thread 1: 2
Thread 2: 1
Thread loop Main: 2
Thread loop Main: 3
Thread 2: 2
Thread loop Main: 4
Thread 3: 0
```

Thread 3: 1

Wenn Sie sicherstellen wollen, dass Ihr Thread vollständig ausgeführt wird, probieren Sie die Funktion `handle.join` aus. Verschieben Sie es auch oder verwenden Sie mal mehrere `JoinHandle`.

151 Beispiel Sourcecode über mehreren Dateien :

Main.rs

```
mod another_sourcefile;
fn main() {
    let person: another_sourcefile::Person =
another_sourcefile::get_default_person();
    println!("Person name: {}", person.name);
    println!("Person age: {}", person.age);
    println!("");
    println!("Print someString: {}", another_sourcefile::get_some_string());
    println!("\n");
    println!("\nPrint meine Funktion: {}",
another_sourcefile::meine_funktion(32,12));
}
```

another_sourcefile.rs

```
pub struct Person {
    pub name : String,
    pub age : u8,
}
pub fn get_default_person() -> Person {
    return Person {
        name: "Mr. Nobody".to_string(),
        age: 77,
    };
}
pub fn get_some_string() -> String {
    return String::from("some string");
}
pub fn meine_funktion(z:i32, ypsilon:i32) -> i32{
    println!("..in der Funktion...{}", z);
    println!("..ypsilon in Funktion...{}", ypsilon);
    let t:i32 = z + ypsilon;
    return t;
}
```

Ausgabe:

```
Person name: Mr. Nobody
Person age: Mr. Nobody

Print someString: some string

..in der Funktion...32
..ypsilon in Funktion...12

Print meine Funktion: 44
```

B.

152 Beispiel Void als Rückgabe einer Funktion :

```
fn clearscreen() -> () { // () bedeutet void
    println!("\n\n\n\n\n\n\n\n\n\n");
}
fn main() {
    println!("start");
    clearscreen();
    println!("stop");
}
```

Ausgabe:

start

stop

Text

153 Beispiel keypress mit break :

main.rs

```
use console::Term;

fn main() {
    let buffered_stdout = Term::buffered_stdout();

    'keypress_loop: loop {
        if let Ok(pressed_key) = buffered_stdout.read_char() {
            match pressed_key {
                'i' => method_call(String::from("I")),
                'm' => method_call(String::from("M")),
                'j' => method_call(String::from("J")),
                'k' => method_call(String::from("K")),
                'q' => break 'keypress_loop',
                _ => method_call(String::from("useless")),
            }
        }
    }
}

fn method_call(mystring: String) -> () {
    println!("pressed: {}", mystring);
}
```

Cargo.toml

```
[package]
name = "keypress_console_term"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
console = "0.15.8"
```

Ausgabe:

```
pressed: I
pressed: J
pressed: K
pressed: useless
pressed: useless
```

Process finished with exit code 0

B.

154 Beispiel keypress mit clearscreen :

main.rs

```
use console::Term;

use std::io::{stdout, Write};
use crossterm::{terminal::{ClearType, Clear}, QueueableCommand, cursor::{MoveTo, Hide}};

fn main() {
    let buffered_stdout = Term::buffered_stdout();

    'keypress_loop: loop {
        if let Ok(pressed_key) = buffered_stdout.read_char() {
            match pressed_key {
                'i' => method_call(String::from("I")),
                'm' => method_call(String::from("M")),
                'j' => method_call(String::from("J")),
                'k' => method_call(String::from("K")),
                'q' => break 'keypress_loop',
                'c' => clear_screen(),
                _ => method_call(String::from("useless")),
            }
        }
    }
}

fn method_call(mystring: String) -> () {
    println!("pressed: {}", mystring);
}

pub fn clear_screen() {
    let mut out = stdout();
    out.queue(Hide).unwrap();
    out.queue(Clear(ClearType::All)).unwrap();
    out.queue(MoveTo(0, 0)).unwrap();
    out.flush().unwrap();
}
```

cargo.toml

```
[package]
name = "keypress_console_term"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
console = "0.15.8"
crossterm = "0.16.0"
}
```

Ausgabe:

```
pressed: I
pressed: J
pressed: K
```

pressed: useless
pressed: useless

Process finished with exit code 0

155 Beispiel für das Senden und Empfangen von Daten mittels Channel :

main.rs

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {

    let mut count = 0u32;

    let (sender, receiver) = channel();

    let _my_join_handle = thread::spawn(move || {
        let mut my_vec = vec![];
        loop {
            if let Ok(_msg) = receiver.try_recv() {

                my_vec.push(_msg);
                println!("OK");
                println!("msg {}", my_vec.last().unwrap());
            }

            if my_vec.len() % 100 == 0{
                println!("count: {}", my_vec.len());
            }
        }
    });

    loop {
        count += 1;
        let my_string: String = count.to_string();
        println!("sending {}", my_string);
        sender.send(my_string).unwrap();
        if count == 101000 { // u32::MAX
            return;
        }
    }
}
```

cargo.toml

```
[package]
name = "keypress_example"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

Ausgabe:

```
..
msg 55145
OK
msg 55146
OK
```

msg 55147
OK
msg 55148

Process finished with exit code 0

156 Beispiel für eine unendliche Schleife die durch if-Abfrage beendet wird :

main.rs

```
use std::time::Duration;
use std::thread::sleep;

fn main() {

    let mut count = 0;

    loop {
        count += 1;
        print!("{}", count);
        sleep(Duration::from_millis(5));
        if count == 2000{
            break;
        }
    }
}
```

cargo.toml

```
[package]
name = "thread_example"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

Ausgabe:

```
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,...
```


157 Einlesen der zuletzt modifizierten Datei eines Verzeichnisses :

main.rs

```
use std::fs::read_dir;

fn main() {

    let last_modified_file_from_directory = read_dir("/home/tkalman/resources")
        .expect("Directory not findable")
        .flatten()
        .filter(|f| f.metadata().unwrap().is_file())
        .max_by_key(
            |mbk| mbk.metadata()
                .unwrap()
                .modified()
                .unwrap()
        );

    let file_name: String ;
    match last_modified_file_from_directory {
        Some(ref last_file) => {
            file_name = last_file.path().to_str().unwrap().to_string();
        }
        None => {
            println!("Error reading DirEntry");
            file_name = String::from("no file found");
        },
    },
}

println!("string from dir_entry: {}" , file_name);
}
```

cargo.toml

```
[package]
name = "line_test"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

Ausgabe:

```
string from dir_entry: /home/tkalman/resources/Textdatei (1).txt
```

158 Beispiel XXX :

```
fn main() {
```

Ausgabe:

```
z
```

159 Beispiel XXX :

```
fn main() {
```

Ausgabe:

```
z
```

Tipps

Laden von größeren Dateien:

Möchte man z.B. grössere Dateien laden, dann sollte man das Makro `include_str!` verwenden

```
let grosse_datei = include_str!("irgendeinegrossedatei.txt");
```

Stichworte

_variablen_name: Wenn man ungenutzte Variablen verwenden möchte, kann man ein Unterstrich davor stellen.

::<T> Turbofisch

Argumente: Die konkreten Werte der Parameter die beim Aufruf einer Funktion an der aufrufenden Stelle mitgegeben werden, nennt man Argumente.

Box<T> :

Werte werden auf dem Heap zugewiesen. Es zeigt damit auf Daten im Heap (Haldenspeicher). Der Zeiger selbst ist allerdings auf dem Stack (Stapel Speicher). Wird verwendet, wenn zur Laufzeit die Grösse eines Objekts nicht bekannt ist.

Character: Einzelne Buchstaben einer Zeichenkette werden Character genannt. Ihre Grösse ist in Rust immer 4 Byte lang. Da es sich um UTF-8 Character handelt.

Closure:

Anonyme Funktion. Lässt sich einer Variablen zuweisen. Kann Funktionen übergeben werden.

Compound Typen oder Verbund Typen:

ein oder mehrere Werte können in Arrays (immer gleicher Typ) oder Tuple (unterschiedliche Typen möglich) abgelegt werden.

Data Memory (Datenspeicher): Daten die permanent zur Laufzeit eines Programms zur Verfügung stehen müssen. Sie sind für statische Daten und Daten mit fester Grösse.

Escaping Character:

\n neue Zeile
\t Tabulator
\0 null
\r Carriage Return
\' Einfaches Hochkomma
\" Doppeltes Hochkomma
\\ Backslash

Z.B. zur Angabe in einem String Literal:

```
let mein_string: &'static str = "\"Text\"";
```

Heap Memory: Dynamische Speichermanagement für Daten die in Ihrer Größe veränderbar sind. Speicher kann angefordert und gelöscht werden (Allocation und Deallocation). Die Organisation dieses Speichers ist am aufwändigsten und dauert oft länger als die Verwaltung der anderen Speicher.

Lebensdauer:

In Rust müssen wir teilweise extra Lebensdauer angeben, damit Variablen auch dann gültig sind, wenn wir sie verwenden.

Parameter: die in der Funktion formulierten Variablen zwischen den ()-Klammern nennt man Parameter einer Funktion.

Skalare Typen:

Einzelne Werte wie Ganzzahlen, Fließkommazahlen, Character oder Boolean.

Stack Speicher: Wenn Variablen innerhalb von Funktionen angelegt werden, werden diese hier abgelegt und beim Verlassen der Funktion wieder automatisch gelöscht.

String vs &str:

Sie sollten einen String verwenden, wenn sobald man Zeichenkette verändern möchte. Er wird auf dem Heap angelegt.

&str, sollte man verwenden, wenn Sie die Zeichenkette nur lesend verwenden wollen.

Sie können es sich wie unter Java so vorstellen, dass der String in Rust dem StringBuilder entspricht und der String in Java entspricht dem &str. In Java sollte man nicht Strings mit Strings konkatenieren, wenn es sich um mehrere Operationen über mehrere Zeilen handelt, dafür sollte man dort den Stringbuilder verwenden.

Quelle: <https://stackoverflow.com/questions/24158114/what-are-the-differences-between-rusts-string-and-str>

Traits: sind z.B. mit Interfaces in Java vergleichbar.

Variablen: sollten immer snake_case sein, auch Dateinamen.

Anhang:

160 Beispiel Zahlenratespiel:

```
use std::cmp::Ordering;
use std::io;
use rand::Rng;

fn main() {
    println!("Zahlenraten zwischen 1 und 10!");

    let gesuchte_zahl_als_int: u32 = rand::thread_rng().gen_range(1..=10);
    let mut collected_numbers: Vec<u32> = Vec::with_capacity(10);

    loop {
        println!("\n\r");
        println!("Bitte gib eine Zahl zwischen 1 und 10 ein.");

        let mut number_to_guess_as_string = String::new();
        io::stdin()
            .read_line(&mut number_to_guess_as_string)
            .expect("Fehler bei der Eingabe");

        let number_to_guess_as_int: u32 = match number_to_guess_as_string.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        collected_numbers.push(number_to_guess_as_int);
        collected_numbers.sort();

        for cn in &collected_numbers {
            println!("bereits eingegeben: {}", cn);
        }

        println!("Gerade eingegeben: {}", number_to_guess_as_string);

        match number_to_guess_as_int.cmp(&gesuchte_zahl_als_int) {
            Ordering::Less | Ordering::Greater => println!("nicht die Zahl die wir suchen"),
            Ordering::Equal => {
                println!("!!!!!!!!!!!!Gefunden!!!!!!!!!!!!");
                break;
            }
        }
    }
}
```

Ausgabe:

```
Zahlenraten zwischen 1 und 10!
Bitte gib eine Zahl zwischen 1 und 10 ein.

4

Bitte gib eine Zahl zwischen 1 und 10 ein.
```

bereits eingegeben: 4
Gerade eingegeben: 4

nicht die Zahl die wir suchen

Bitte gib eine Zahl zwischen 1 und 10 ein.
5

bereits eingegeben: 4
bereits eingegeben: 5
Gerade eingegeben: 5

nicht die Zahl die wir suchen

Bitte gib eine Zahl zwischen 1 und 10 ein.
6

bereits eingegeben: 4
bereits eingegeben: 5
bereits eingegeben: 6
Gerade eingegeben: 6

nicht die Zahl die wir suchen

Bitte gib eine Zahl zwischen 1 und 10 ein.
7

bereits eingegeben: 4
bereits eingegeben: 5
bereits eingegeben: 6
bereits eingegeben: 7
Gerade eingegeben: 7

!!!!!!!Gefunden!!!!!!!

Diese Zahlenratespiel gibt keinen Hinweis darauf, ob wir uns über oder unter dem gesuchten Wert befinden, dafür listet es die bereits gesuchten Zahlen auf. Erweitern Sie das Spiel so, dass es Hinweise gibt, ob sich der Suchende über oder unter dem Wert befindet.