

11.

Generics & Traits

Generics

- Funktion oder Datentyp soll mit mehreren Typen funktionieren
- „Liste von Dingen“ allgemein implementieren
 - Nicht „Liste von **i32**s“, „Liste von **bool**ans“, ... separat implementieren

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



```
let a = Some(3);    // : Option<i32>  
let b = Some(true); // : Option<bool>  
  
let x: Option = Some(3); // error
```



Ist „**Option**“ ein Typ?
Wenn nein, was dann?



Später!

Syntax

- *Erst*: Deklaration Typparameter
- *Dann*: Benutzung im Rumpf
- Platzhalter für tatsächlichen Typ
- Name: Meist ein Großbuchstabe
 - Üblich: **T** für „**t**ype“
 - Wenn beserer Name nötig: **CamelCase**

Deklaration

Structs

```
struct Foo<T> {  
    field: T,  
}
```

Benutzung

Deklaration

Enums

```
enum Foo<T> {  
    Variant(T),  
}
```

Benutzung

impl Blöcke

```
impl<T> Foo<T> { ... }
```

Deklaration

Benutzung

Funktionen

```
fn foo<T>(x: T) { ... }
```

Deklaration

Benutzung

Tuple Structs

```
struct Foo<T>(T);
```

Deklaration

Benutzung



Bis auf impl-Block Parameterliste immer nach Namen!

impl-Block

Benutzung

```
impl<T> Option<T> {  
    fn unwrap(self) -> T {  
        match self {  
            Some(t) => t,  
            None => panic!(),  
        }  
    }  
}
```

```
impl Option<i32> {  
    fn maybe_increment(&mut self) {  
        if let Some(ref mut t) = *self {  
            t += 1;  
        }  
    }  
}
```

- Typparameter in ganzem impl-Block nutzbar
- Auch für speziellen Typen möglich

Mehrere Parameter/Deklarationen

```
impl<T> Option<T> {  
    fn ok_or<E>(self, err: E) -> Result<T, E> {  
        match self {  
            Some(t) => Ok(t),  
            None => Err(err),  
        }  
    }  
}
```

```
// Two type parameters  
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

- Zusätzliche Deklaration von Typparametern an Funktion/Methode möglich
- Deklaration mehrere Parameter mit Komma getrennt

Typinferenz und Turbofish

```
// type inference works 😊  
let o = Some(3);  
let r = o.ok_or(true);  
  
// types of `o` and `r`?  
// o: Option<i32>  
// r: Result<i32, bool>
```

- Typannotation meist nicht nötig
- *Sonst*: Typparameter explizit mit Turbofish angeben
 - „Rust’s ugliest syntax yet“


```
// (in std::mem)  
/// Returns the size of `T` in bytes  
fn size_of<T>() -> usize {  
    // compiler magic  
}  
  
// ehm... size of what?  
let size = size_of();  
  
// use turbofish ::<>  
let size = size_of::
```

Typinferenz bei Enums

```
// error: unable to infer enough  
// type information  
let a = None;  
  
fn bind_port(port: Option<u16>) { ... }  
  
// compiler knows: Option<_>  
// where „_“ = „something“  
let b = None;  
  
// Aha! `b` should have been  
// Option<u16> all along!  
bind_port(b);
```

- Typ kann erst später durch Benutzung der Variable inferiert werden
- Compiler benutzt alle verfügbaren Informationen

```
// expected type `Option<u16>`  
// found type `Result<bool, _>`  
bind_port(Ok(false));
```



Noch nicht klar

Mehr Beispiele

```
let arr = [1, 2, 3];
let a = None;

// arr.len() is use, therefore
// `a` has to be Option<usize>
for i in a.unwrap() .. arr.len() {
    println!("{}", i);
}

// return types influence inference
fn foo() -> Vec<f64> {
    Vec::new()
}
```

```
fn parse<T>(s: &str) -> Result<T, ?> {
    // we will be able to
    // understand this later ...
}
```

```
// Type annotations on the left side
// work, too!
let x: i32 = "27".parse().unwrap();
let x = "27".parse::<i32>().unwrap();

// Partial type
let x: Result<i32, _> = "27".parse();
```


Mal etwas ausprobieren...

```
// We want it to work for multiple  
// types, not just `i32`.  
/// Returns the smaller element.  
fn min<T>(a: T, b: T) -> T {  
    if a < b { a } else { b }  
}  
  
// error: binary operation `<`  
// cannot be applied to type `T`  
//  
// note: an implementation of  
// `std::cmp::PartialOrd` might  
// be missing for `T`
```

- Wir wissen nichts über den generischen Parameter¹
- „Fähigkeiten“ müssen explizit angefordert werden

 Hallo *Traits*!

¹ Wir wissen, dass sie **Sized** sind. Später mehr.

Trait Bounds

```
// We want it to work for multiple  
// types, not just `i32`.  
/// Returns the smaller element.  
fn min<T>(a: T, b: T) -> T  
    where T: PartialOrd, ←  
{  
    if a < b { a } else { b }  
}  
  
// Trait bounds can also be specified  
// inline (only for usage with simple  
// and short trait bounds!)  
fn min<T: PartialOrd>(…) -> T { … }
```

- Mit „Trait Bounds“ Fähigkeiten des Typs verlangen
- Schränkt die Menge möglicher Typen ein
- **PartialOrd**: „Typ ist vergleichbar“, später mehr

Traits definieren und implementieren

```
trait Speak {  
    fn speak(&self);  
}  
  
struct Cat;  
impl Speak for Cat {  
    fn speak(&self) {  
        println!("meow");  
    }  
}
```

- Traits als Interfaces
- Können implementiert werden

```
struct Pokemon { name: String }  
impl Speak for Pokemon {  
    fn speak(&self) {  
        println!("{}", self.name);  
    }  
}  
  
let cat = Cat;  
let poki = Pokemon {  
    name: "peter".into(),  
};  
  
// Use like regular methods  
cat.speak();    // meow  
poki.speak();  // peter
```

Trait Bounds an Funktionen

```
trait Speak {  
    fn speak(&self);  
}  
  
fn foo<T>(x: &T) {  
    x.speak(); // error  
}  
  
fn bar<T>(x: &T)  
    where T: Speak ←  
{  
    x.speak(); // works  
}  
  
bar(&cat); // works
```

```
let x = 3;  
bar(&x); // error
```

- Anforderung an Typen im Funktionskopf festgelegt
- Fehler bei Nichterfüllung beim Aufruf der Funktion
 - Im Gegensatz zu Template-Fehlern :-o
- Mehrere Bounds mit **T: A + B** ↓

Trait Definition

```
trait <Name> {  
    // They lack a function body and  
    // their implementation has to  
    // be provided by the implementing  
    // type.  
    fn <required_method>(...);  
  
    // Default methods already provide  
    // a body, but they can be overridden  
    // in a type's implementation.  
    fn <default_method>(...) { ... }  
  
    // A type the implementation has to  
    // provide (more later)  
    type <AssociatedType>;  
}
```

- **Required Methods:**
Jeder implementierender Typ muss Rumpf bereitstellen.
- **Default Methods:**
Methoden Rumpf schon vorhanden, kann aber überschrieben werden.
- **Associated Type:**
Typ, der von der Implementation bereitgestellt werden muss (später mehr)

Regeln für Trait-Nutzung

```
mod foo {  
    trait Speak { fn speak(&self); }  
  
    struct Cat;  
    impl Speak for Cat { ... }  
}  
  
fn main() {  
    use foo::Speak;  
  
    let cat = foo::Cat;  
    cat.speak();  
}
```

- Trait in Scope (mit **use**)
— oder —
- Universal Function Call Syntax
 - (eher selten!)

error: no method named `speak` found for type `foo::Cat` in the current scope
--> type.rs:15:9

```
15 |         cat.speak();  
    |                ^^^^^
```

= help: items from traits can only be used if the trait is in scope; the following trait is implemented but not in scope, perhaps add a `use` for it:
= help: candidate #1: `use foo::Speak`

Universal Function Call Syntax

```
impl Cat {  
    fn attack(&self, strong: bool) { ... }  
}  
  
let cat = Cat;  
cat.attack(true);  
Cat::attack(&cat, true); ←
```

```
trait Speak { fn speak(&self); }  
impl Speak for Cat { ... }  
  
let cat = Cat;  
Speak::speak(&cat); ←
```

- Punkt-Syntax ist Zucker
- **UFCS**: Explizite Form
- **self** Parameter wird explizit übergeben

- Gut für:
 - Disambiguierung
 - Methode als *Funktionspointer*

Universal Function Call Syntax

```
impl Cat {  
    fn bar() -> u64 { 42 }  
}  
  
trait Foo { fn bar() -> u64; }  
impl Foo for Cat {  
    fn bar() -> u64 { 27 }  
}  
impl Foo for Dog { ... } // more impls  
  
assert_eq!(42, Cat::bar());  
  
// What impl should be chosen?!  
Foo::bar(); // error
```

- Maximal-explizite Syntax:
`<Type as Trait>::method(...)`
- In seltenen Fällen nötig

```
assert_eq!(27, <Cat as Foo>::bar());
```



Regeln für Implementierungen

```
impl <Type> { ... }
```

- **<Type>** muss in der jetzigen Crate definiert sein
 - Verhindert `impl i32 { ... }`

```
impl <Trait> for <Type> { ... }
```

- **<Type>** oder **<Trait>** muss in der jetzigen Crate definiert sein
 - Ziel: Mehr Stabilität durch Berechenbarkeit
 - „Orphan Rules“

Beispiel: Formatting Traits


in `std::fmt`

```
pub trait Display {
    fn fmt(&self, &mut Formatter)
        -> Result<(), Error>;
}

pub trait Debug {
    fn fmt(&self, &mut Formatter)
        -> Result<(), Error>;
}
```

```
fn print_twice<T>(x: T) {
    println!("{}", x); // error
    println!("{}", x); // error
}

fn print_twice<T: fmt::Display>(x: T) {
    ...
}
```



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

```
impl fmt::Display for Point { // this is the f64-Point!
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error> {
        write!(f, "[{}, {}]", self.x, self.y)
    }
}
```

```
let p = Point::origin();
println!("{}", p);
```

Beispiel: Formatting Traits

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

```
struct GenPoint<T> {  
    x: T, y: T,  
}
```

```
impl<T: fmt::Display> fmt::Display for GenPoint<T> {  
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error> {  
        write!(f, "[{}, {}]", self.x, self.y)  
    }  
}
```

```
/// Prints a value as user-faced output and  
/// as debug output  
fn print_both<T>(x: &T)  
    where T: fmt::Display + fmt::Debug  
{  
    println!("{}", x);  
    println!("{:?}", x);  
}
```

Derivable Traits

- `#[derive(...)]` Attribut generiert impl-Block
- Manchmal ist manuelle Implementation nötig
- Wenn Typparameter: in impl-Block mit Trait Bound
 - Manchmal problematisch

```
#[derive(Debug)]  
struct Point { x: f64, y: f64 }
```

generiert

```
struct Point { x: f64, y: f64 }  
  
impl fmt::Debug for Point { ... }
```

```
#[derive(Debug)]  
struct GenPoint<T> { x: T, y: T }
```

generiert

```
struct GenPoint<T> { x: T, y: T }  
  
impl<T> fmt::Debug for GenPoint<T>  
    where T: fmt::Debug  
{ ... }
```

Derivable Traits

- Vergleich-Traits: **PartialEq**, **Eq**, **PartialOrd**, **Ord**
- **Clone** und **Copy** (schon bekannt)
- **Hash**: Hashwert einer Instanz kann berechnet werden
- **Default**: Eine Standardinstanz kann erstellt werden
- **Debug** (schon bekannt)

- Funktioniert nur wenn Felder schon Trait implementieren!
- Eigene derivable Traits via Compiler Plugin (später mehr!)

Beispiel: Clone & Copy

```
/// A common trait for the ability to explicitly
/// duplicate an object.
trait Clone {
    /// Returns a copy of the value.
    fn clone(&self) -> Self;

    /// Performs copy-assignment from `source`.
    fn clone_from(&mut self, source: &Self) { ... }
}
```

```
/// Types whose values can be duplicated
/// simply by copying bits.
trait Copy: Clone {}
```

Hä?

- Leerer Rumpf: *Marker Trait*
 - Markieren nur Eigenschaft
- Doppelpunktsyntax?

Trait „Inheritance“

```
trait Foo: RequiredA + RequiredB { ... }
```

- Voraussetzung für implementierende Typen
 - „Alle Typen die mich implementieren, müssen auch diese anderen Traits implementieren“
- Sinnvoll für:
 - Zusammenfassung mehrere Traits
 - „kann X und Y und Z!“
 - Benötigt für Rumpf von Default-Methoden
- Nicht wirklich die „Vererbung“ aus OOP!

```
trait Extra: Base {}  
trait Base {}  
  
// error: the trait  
// bound `i32: Base` is  
// not satisfied  
impl Extra for i32 {}
```

Beispiel: PartialEq und Eq

(nicht Original-Code aus std!)

```
/// Trait for equality comparisons
trait PartialEq {
    /// This method tests for `self` and `other` values
    /// to be equal, and is used by `==`.
    fn eq(&self, other: &Self) -> bool;

    /// This method tests for `!=`.
    fn ne(&self, other: &Self) -> bool { ... }
}
```

Default Methode

- Schränkt dieses Trait zu sehr ein?
→ Vergleich nur mit gleichem Typen möglich! ☹️

Beispiel: PartialEq und Eq

```
/// Trait for equality comparisons
trait PartialEq<Rhs = Self> {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { ... }
}
```

(immer noch nicht ganz
Original-Code aus std!)

- Traits können auch Typparameter haben
- Syntax „Parameter = Typ“ bedeutet: Default
 - Wenn kein Parameter spezifiziert wird → Defaulttyp
 - Funktioniert nicht nur für Traits

```
impl PartialEq for String { ... }
impl PartialEq<str> for String { ... }
```

Partial{Eq, Ord} vs. {Eq, Ord}?

- PartialEq: „Partielle Äquivalenzrelation“
 - Symmetrisch und transitiv
- Eq: (volle) Äquivalenzrelation
 - Zusätzlich reflexiv
- PartialOrd: Vergleichbare Typen
 - Antisymmetrisch und transitiv
- Ord: Typen mit linearer Ordnung („total order“)
 - Zusätzlich total

```
trait Eq: PartialEq<Self> {}
```

```
pub trait Ord: Eq + PartialOrd<Self> {  
    fn cmp(&self, other: &Self)  
        -> Ordering;  
}
```

`f64` & `f32` sind `Partial{Eq, Ord}`
aber nicht `{Eq, Ord}`!

```
trait PartialOrd<Rhs = Self>: PartialEq<Rhs> {  
    fn partial_cmp(&self, other: &Rhs)  
        -> Option<Ordering>;  
}
```

Beispiel: Addition

```
/// This trait is used to specify the functionality of `+`.  
pub trait Add<Rhs = Self> {  
    type Output; ← -----  
    fn add(self, rhs: Rhs) -> Self::Output;  
}
```

Assoziierter Typ

- Assoziierter Typ:
 - Implementierender Typ darf assoz. Typ festlegen

```
impl Add for Point {  
    type Output = Point;  
    fn add(self, rhs: Point) -> Self::Output { ... }  
}
```

Operatorüberladung

- Operatoren mit eigenen Typen nutzen (z.B. Vektoren u. Matrizen)
 - C++ ermöglicht es auch, Java nicht
- In Rust: Traits in [std::ops](#) (u.a.) implementieren

a + b



::std::ops::Add(a, b)

Beispiel

```
let p = Point::new(3, 5);  
let x = p + Point::origin();  
  
println!("{}", x);
```

Beispiel: Into/From

```
/// Construct Self via a conversion.  
trait From<T> {  
    fn from(T) -> Self;  
}
```

```
/// A conversion that consumes self.  
trait Into<T> {  
    fn into(self) -> T;  
}
```

- Für eindeutige, verlustfreie, fehlerlose Konvertierungen
 - `&str` zu `String`, `u8` zu `u16`, ...
- Selber lieber **From** implementieren

```
// Blanket impls  
impl<T> From<T> for T { ... }  
impl<T, U> Into<U> for T where U: From<T> { ... }
```

Sized Trait

```
trait Sized {}
```

- *Marker Trait* (besondere Bedeutung nur durch Compiler)
- „Hat Typ feste Größe zur Kompilierzeit?“
- Wird automatisch für (fast) alle Typen implementiert
- Unsized Typen:
 - Slices: **[T]** und **str**
 - *Traits Objects* (später mehr)
- Typparameter haben impliziten Bound auf **Sized**
 - Aufheben mit **?Sized**

Ein letztes Mal: PartialEq

```
/// Trait for equality comparisons
trait PartialEq<Rhs = Self> {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { ... }
}
```

```
impl PartialEq for String { ... }
impl PartialEq<str> for String { ... }
```

Error!

```
/// Trait for equality comparisons
trait PartialEq<Rhs = Self>
    where Rhs: ?Sized
{
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { ... }
}
```

Impliziten Trait Bound
aufheben
→ Mehr Typen zulassen

Extension Traits

- Vorhandene Typen oder Traits erweitern
- **Ext** am Ende des Namens

```
trait BaseTenExt {  
    fn hundred(self) -> Self;  
}  
  
impl BaseTenExt for i32 {  
    fn hundred(self) -> Self { self * 100 }  
}  
  
let x = 3.hundred();  
println!("{}", x);
```


Wichtige Traits

- Die bisher besprochenen...
- **Default**: Sinnvolle Standardinstanz
- **std::io** Traits
 - Read
 - Write
 - Seek
- **Send** und **Sync**: Später mehr
- **Drop**: Später mehr
- **std::iter** Traits: Jetzt mehr...

```
trait Default: Sized {  
    fn default() -> Self;  
}
```

Iteratoren

```
trait Iterator {  
    type Item;  
    fn next(&mut self) -> Option<Self::Item>;  
  
    // *many* default methods  
}
```

- **Iterator:**
 - Kann Element liefern
 - Wenn kein Element vorhanden ist: **None**
 - Viele Hilfsfunktionen; die meisten erst mit Closures sinnvoll → Später!
- Sehr allgemeine Definition → Findet viel Anwendung
- Erweiterungen (setzen **Iterator** voraus)
 - **ExactSizeIterator**: Weiß, wie viel Elemente geliefert werden; **len()**
 - **DoubleEndedIterator**: Kann Elemente vom Ende liefern; **next_back()**

IntoIterator


```
pub trait IntoIterator {  
    type Item;  
    type IntoIter: Iterator<Item=Self::Item>;  
    fn into_iter(self) -> Self::IntoIter;  
}
```

- „In Iterator konvertierbar“
- Voraussetzung für **for**-Schleife!
- Beispiel: **Vec<T>**
 - Implementiert *nicht* **Iterator** (merkt sich Position nicht)
 - Implementiert aber **IntoIterator**

```
// Blanket impl  
impl<I> IntoIterator for I  
    where I: Iterator  
{ ... }
```

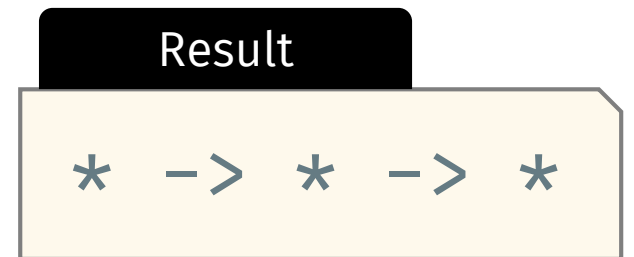
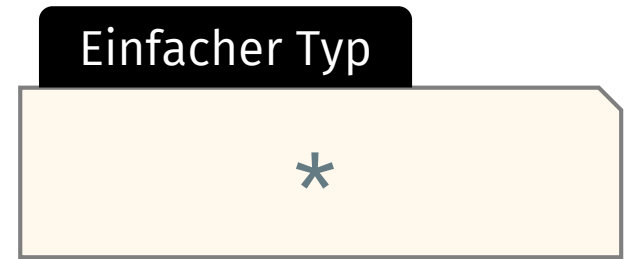
```
impl<T> IntoIterator for Vec<T> {  
    type Item = T;  
    ...  
}
```

```
impl<T> IntoIterator for &Vec<T> {  
    type Item = &T;  
    ...  
}
```



Nochmal: Typtheorie :3

- `i32` ist ein Typ
 - `Option<i32>` ist ein Typ
 - Was ist `Option`?
- ➔ Typkonstruktor
- Bekommt einen Typen, „returned“ einen Typen
 - Typparameter in Rust kann nur konkrete Typen akzeptieren (sonst: „*higher kinded types*“)



↑
„Kind“