Seminararbeit Traits und Enums in Rust

Mario Occhinegro
HKA University of Applied Sciences

Inhaltsverzeichnis

1 Einleitung **1**

2 Enums **2**
2.1 Enums in Rust . . . 2
2.1.1 Normale Enums . . . 2
2.1.2 Enum mit Werten . . . 2
2.1.3 Enum mit Funktionen . . . 3
2.2 Enums in Java . . . 3
2.2.1 Normale Enums . . . 4
2.2.2 Enums mit Werten . . . 4
2.2.3 Enum mit Funktionen . . . 4
2.3 Mächtigkeit von Rust Enums . . . 4
2.3.1 Der Enum als algebraischer Datentyp . . . 4
2.3.2 Generische Enums . . . 5
2.3.3 Rust Enums und die Vermeidung von Nullpointer-Ausnahmen 5
2.3.4 Rekursive Enums . . . 6
2.3.5 Match Statement . . . 6
2.3.6 Feste Enum Cases . . . 7
2.3.7 Nested Pattern Matching . . . 9
2.3.8 Erweiterbare Funktionen für Enums . . . 9
2.4 Rust-Enum-Funktionalität in Java . . . 10
2.4.1 Switch Case vs Match . . . 10
2.4.2 Expression-Logik in Java . . . 10
2.4.3 Java Enums am Limit . . . 12

3 Traits **13**
3.1 Traits in Rust . . . 13
3.1.1 Einfacher Trait . . . 13
3.1.2 Shorthand Schreibweise . . . 14
3.1.3 Default-Implementationen . . . 14
3.1.4 Trait Bounds . . . 15
3.1.5 Multiples Binding . . . 15
3.1.6 Dynamische Traits . . . 15
3.1.7 Kurzschreibweise für dynamische Traits . . . 15
3.1.8 Platzhaltertypen . . . 15
3.1.9 Assoziierte Konstanten . . . 16
3.1.10 Supertraits . . . 16
3.2 Mächtigkeit von Traits . . . 17
3.2.1 Gleiche Methodensignatur . . . 17
3.2.2 Generische Mehrfachimplementierung . . . 18
3.2.3 Referenzierung des eigenen Typen . . . 19
3.2.4 Funktionalität für Third-Party-Datentypen . . . 19
3.2.5 Referenzierung des eigenen Typen . . . 20
3.2.6 Konditionelle Implementierung . . . 20
3.3 Traitfunktionalität in Java . . . 20
3.3.1 Gleiche Methodensignatur . . . 20
3.3.2 Generische Mehrfachimplementierung . . . 21
3.3.3 Funktionalität für Third-Party-Datentypen . . . 23

3.3.4 Referenzierung des eigenen Typens 23
3.3.5 Konditionelle Implementierung 24

4 Vergleich **25**
4.1 Enums 25
4.2 Traits 25

Zusammenfassung

1 Einleitung

In der Welt der Programmiersprachen gibt es eine Vielzahl von Ansätzen, um Datenstrukturen zu modellieren und die Funktionalität zwischen verschiedenen Typen zu teilen. In diesem Vergleich werfen wir einen Blick auf zwei mächtige Konzepte: Enums und Traits in Rust im Vergleich zu den Möglichkeiten, die Java bietet. Rust, eine moderne und stark typisierte Programmiersprache, bietet mit Enums algebraische Datentypen, die es ermöglichen, eine begrenzte Anzahl von Varianten zu definieren. Im Gegensatz zu herkömmlichen Enumerationen in anderen Sprachen sind Rust Enums jedoch viel flexibler. Sie können komplexe Datenstrukturen darstellen und sind nicht auf einfache numerische Werte beschränkt. Ein bemerkenswertes Merkmal von Rust Enums ist das sogenannte Pattern Matching. Hierbei handelt es sich um einen Mechanismus, der sicherstellt, dass alle möglichen Fälle des Enums abgedeckt sind. Dies erhöht die Sicherheit des Codes, da der Compiler sicherstellt, dass keine unbehandelten Fälle auftreten können. Im Vergleich dazu bietet Java keine eingebaute Unterstützung für algebraische Datentypen oder Pattern Matching, was zu einer potenziell fehleranfälligeren Handhabung von Varianten führen kann. Ein weiteres mächtiges Konzept in Rust sind Traits. Diese können als Mengen über einem Typ betrachtet werden und ähneln in gewisser Weise den Interfaces und Vererbungskonzepten in Java. Durch die Verwendung von Traits können Funktionalitäten zwischen verschiedenen Datentypen geteilt werden, wodurch Code modularer und wiederverwendbarer wird. Traits dienen auch als Prädikate über einem Typen, indem sie bestimmte Verhaltensweisen oder Fähigkeiten definieren, die ein Typ haben muss, um ein bestimmtes Trait zu implementieren. Dies ermöglicht eine flexible Komposition von Funktionen und Verhalten, ohne eine starre Vererbungshierarchie zu erzwingen. Im weiteren Verlauf dieses Vergleichs werden wir detaillierter auf die Vorteile von Traits gegenüber Interfaces und Vererbung eingehen. Darüber hinaus werden wir prüfen, ob Java überhaupt in der Lage ist, die gleiche Funktionalität wie Rust zu bieten und welche Kompromisse möglicherweise gemacht werden müssen. Mit diesem Hintergrundwissen tauchen wir nun in den Vergleich der Enums und Traits in Rust mit den Möglichkeiten in Java ein, um die Stärken und potenziellen Unterschiede zwischen diesen beiden Sprachen genauer zu untersuchen.

2 Enums

Enums sind in der Programmierung eine Möglichkeit, eine begrenzte Anzahl von Varianten oder Optionen zu definieren. Sie dienen dazu, eine feste Menge von Werten darzustellen, die ein bestimmtes Konzept oder eine bestimmte Entität repräsentieren können. Enums ermöglichen es, den Code lesbarer zu gestalten, indem sie eine klar definierte Liste von möglichen Zuständen oder Varianten zur Verfügung stellen. Auf den ersten Blick mögen Enums in Rust und Java ähnlich aussehen. Beide Sprachen bieten Mechanismen, um eine begrenzte Anzahl von Optionen zu modellieren. Jedoch gibt es einen entscheidenden Unterschied: Rust Enums sind algebraische Datentypen. Algebraische Datentypen in Rust erlauben es, komplexe Datenstrukturen zu definieren, die weit über einfache numerische Werte hinausgehen. Sie können Varianten enthalten, die selbst wiederum Daten oder weitere Optionen beinhalten können. Diese Flexibilität eröffnet ganz neue Möglichkeiten bei der Modellierung von Daten. Ein weiteres starkes Merkmal von Rust Enums ist das abgeschlossene Pattern Matching. Dies bedeutet, dass bei der Verarbeitung eines Enums alle möglichen Fälle explizit abgedeckt werden müssen. Der Compiler erzwingt diese Vollständigkeit, was zu sichererem und fehlerfreiem Code führt. Dadurch wird vermieden, dass unbehandelte Fälle auftreten und potenziell zu Fehlern führen können. Im Gegensatz dazu bieten Java Enums keine eingebaute Unterstützung für algebraische Datentypen oder abgeschlossenes Pattern Matching. Dies kann dazu führen, dass beim Umgang mit Enums in Java unbeabsichtigte Fehler auftreten können, wenn nicht alle Varianten explizit behandelt werden. Durch die Kombination von algebraischen Datentypen und abgeschlossenem Pattern Matching bieten Rust Enums eine überlegene Möglichkeit, komplexe Datenstrukturen zu modellieren und sicherzustellen, dass alle Fälle behandelt werden. Diese Funktionalitäten machen Rust Enums zu einer mächtigen und robusten Technik, die über die Möglichkeiten von Java Enums hinausgeht.

Anhand einfacher Beispiele gehen zeigen ich zunächst, wie simple Enums, Enums mit zugehörigen Werten und Enums mit Funktionen in Rust und Java aussehen. Danach gehe ich auf den konkreten Unterschied ein und zeige den Mehrwert, den Rust Enums bringen. Im Anschluss versuche ich die FUnktionalität, die Rust Enums uns geben in Java (falls es möglich ist) nachzubauen.

2.1 Enums in Rust

2.1.1 Normale Enums

Einfache Auflistungen sind in Rust und Java komplett identisch.

```
enum Animal {
    Dog,
    Cat,
    Bird,
}
```

2.1.2 Enum mit Werten

Wenn wir jedem Enum einen konkreten Wert eines Typs zuordnen möchten, müssen wir das in Rust über eine Funktion machen. Das Pattern Matching auf

das ich nachher noch genauer eingehen werde lässt uns an dieser Stelle bequem und übersichtlich die zugehörigen Werte festlegen. Bei Java ist es möglich, dies einer Option des Enums direkt anzugeben. Dies ist aber kein Nachteil, da es uns erlaubt bei Bedarf weitere Werte hinzuzufügen ohne das Enum zu verändern.

```
enum Animal {
    Dog,
    Cat,
    Bird,
}

impl Animal{
    fn get_label(&self) -> String{
        match self{
            Animal::Dog => String::from("Dog"),
            Animal::Cat => String::from("Cat"),
            Animal::Bird => String::from("Bird"),
        }
    }

    fn get_weight(&self) -> i32{
        match self{
            Animal::Dog => 20,
            Animal::Cat => 10,
            Animal::Bird => 1,
        }
    }
}
```

2.1.3 Enum mit Funktionen

Das definieren von Funktionen über einem Enumtypen funktioniert analog wie mit den hinzufügen von Werten.

```
enum Animal {
    Dog,
    Cat,
    Bird,
}

impl Animal{
    fn is_cat(&self) -> bool{
        match self{
            Animal::Cat => true,
            Animal::Dog => false,
            Animal::Bird => false
        }
    }
}
```

2.2 Enums in Java

- Enums sind spezielle Klasse
- Enumtypen sind Instanzen
- Instanz statisch und final (per default)

2.2.1 Normale Enums

```
enum Animal{
    Dog,
    Cat,
    Bird
}
```

2.2.2 Enums mit Werten

```
enum Animal{
    Dog("Dog", 20),
    Cat("Dog", 10),
    Bird("Bird", 1);

    public final String label;
    public final int weight;

    private Animal(String label, int weight){
        this.label= label;
        this.weight = weight;
    }
}
```

2.2.3 Enum mit Funktionen

```
enum Animal{
    Dog
    Cat
    Bird;

    public boolean isCat(){
        if (this == Animal.Cat){
            return true;
        }else{
            return false;
        }
    }
}
```

2.3 Mächtigkeit von Rust Enums

2.3.1 Der Enum als algebraischer Datentyp

- Algebraische Datentypen

```
fn main() {
    let s1 = Shape::Square(16);
    println!("The area of the shape is {}",s1.area());
}

enum Shape{
    Square(u32),
    Rectangle(u32,u32),
}

impl Shape{
    fn area(&self) -> u32{
        match self {
            Shape::Square(a) => a*a,
            Shape::Rectangle(a,b) => a*b,
        }
    }
}
```

- beliebige Struktur
- werte können sich verändern
- flexibel
- pattern matching lässt uns die einzelnen Werte benutzen

2.3.2 Generische Enums

- Enums können mit generischen Werten generiert werden

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

2.3.3 Rust Enums und die Vermeidung von Nullpointer-Ausnahmen

- Java hat ähnliches Konzept aber mit Klassen
- Nullpointer, der große Milliarden € Fehler

```
mintedfn main() {
    match lookUpAnimal(1){
        Some(Animal::Dog) => println!("Found pet was a dog"),
        Some(_) => println!("Found pet with id 1"),
        None => println!("Sadly no pet was found")
    }
```

```
}

enum Animal{
    Dog,
    Cat,
    Bird,
}

fn lookUpAnimal(id: i32) -> Option<Animal>{
    if(id == 1){
        return Some(Animal::Dog);
    }else{
        return None
    }
}
```

2.3.4 Rekursive Enums

1. Box needed

```
pub enum Exp {
    Int {
        val: i32
    },
    Plus {
        left: Box<Exp>,
        right: Box<Exp>
    },
    Mult{
        left: Box<Exp>,
        right: Box<Exp>
    },
}
```

2.3.5 Match Statement

```
fn main(){
    let e:Exp = Exp::Plus {
        left: Box::new(Exp::Int { val: 10 }), right: Box::new(Exp::Int { val: 22})
    };
    println!("Evaluates to: {}", e.eval());
}

pub enum Exp {
    Int {
        val: i32
    },
    Plus {
        left: Box<Exp>,
        right: Box<Exp>
    },
```

```
    Mult{
        left: Box<Exp>,
        right: Box<Exp>
    },
}

impl Exp{
   fn eval(&self) -> i32{
       match self{
           Exp::Int{val} => *val,
           Exp::Plus{left, right} => left.eval() + right.eval() ,
           Exp::Mult{left, right} => left.eval() * right.eval()
       }
   }
}
```

output

2.3.6 Feste Enum Cases

```
fn main(){
    let e:Exp = Exp::Plus {
        left: Box::new(Exp::Int { val: 10 }), right: Box::new(Exp::Int { val: 22})
    };
    println!("Evaluates to: {}", e.eval());
}
enum Exp {
    Int {
        val: i32
    },
    Plus {
        left: Box<Exp>,
        right: Box<Exp>
    },
    Mult{
        left: Box<Exp>,
        right: Box<Exp>
    },
    Div{
        left: Box<Exp>,
        right: Box<Exp>
    }
}

impl Exp{
   fn eval(&self) -> i32{
       match self{
           Exp::Int{val} => *val,
           Exp::Plus{left, right} => left.eval() + right.eval() ,
```

```
            Exp::Mult{left, right} => left.eval() * right.eval()
        }
    }
}
pub enum Exp {
    Int {
        val: i32
    },
    Plus {
        left: Box<Exp>,
        right: Box<Exp>
    },
    Mult{
        left: Box<Exp>,
        right: Box<Exp>
    },
}

impl Exp{
    fn eval(&self) -> i32{
        match self{
            Exp::Int{val} => *val,
            Exp::Plus{left, right} => left.eval() + right.eval() ,
            Exp::Mult{left, right} => left.eval() * right.eval()
        }
    }
}
```

output

```
error[E0004]: non-exhaustive patterns: `&Exp::Div { .. }` not covered
  --> src/main.rs:27:14
   |
27 |         match self{
   |               ^^^^ pattern `&Exp::Div { .. }` not covered
   |
note: `Exp` defined here
  --> src/main.rs:19:5
   |
7  | pub enum Exp {
   |          ---
...
19 |     Div{
   |     ^^^ not covered
   = note: the matched value is of type `&Exp`
help: ensure that all possible cases are being handled by adding a match arm with a wildca
   |
30 ~            Exp::Mult{left, right} => left.eval() * right.eval(),
31 +            &Exp::Div { .. } => todo!()
   |
```

```
For more information about this error, try `rustc --explain E0004`.
```

2.3.7 Nested Pattern Matching

- kann noch granulareres pattern matching betreiben

```
pub enum Exp {
    Int {
        val: i32
    },
    Plus {
        left: Box<Exp>,
        right: Box<Exp>
    },
    Mult{
        left: Box<Exp>,
        right: Box<Exp>
    },
}

impl Exp{
   fn eval(&self) -> i32{
       match self{
           Exp::Int{val} => *val,
           Exp::Plus{left, right} => left.eval() + right.eval() ,
           Exp::Mult{left, right} =>
            match **left {
                Exp::Int { val:0 } => return 0,
                _ => return left.eval() * right.eval()
            }
       }
   }
}
```

2.3.8 Erweiterbare Funktionen für Enums

```
pub enum Exp {
    Int {
        val: i32
    },
    Plus {
        left: Box<Exp>,
        right: Box<Exp>
    },
    Mult{
        left: Box<Exp>,
        right: Box<Exp>
    },
}
```

```
impl Exp{
   fn eval(&self) -> i32{
       match self{
           Exp::Int{val} => *val,
           Exp::Plus{left, right} => left.eval() + right.eval() ,
           Exp::Mult{left, right} => left.eval() * right.eval()
       }
   }
   fn treeHeight(&self) -> u32 {
                match self{
           Exp::Int{val} => 1,
           Exp::Plus{left, right} => left.treeHeight() + right.treeHeight(),
           Exp::Mult{left, right} => left.treeHeight() + right.treeHeight(),
                }
   }
}
```

2.4 Rust-Enum-Funktionalität in Java

2.4.1 Switch Case vs Match

- Veränderung des Enums spielt für SC keine Rolle
- Dieses Verhalten ist auch nicht in Java über tricks Möglich
- Pattern Matching nur über weitere If/Else Abfragen Möglich
- nested pattern matching nur über weitere If Else Möglich

2.4.2 Expression-Logik in Java

Naiver Ansatz (Geht nicht)

```
public class Expression{
    public static void main(String[] args) {
        Exp p = Exp.Plus;
//not accessible
        System.out.println(p.left);
        System.out.println(p.right);
    }
}

enum Exp {
    Int {
        //cannot be changed(static, final)
        int val;

public int eval() {
            return this.val;
        }

    },
```

```
    Plus {
        Exp left;
        Exp right;

        public int eval() {
            return this.left.eval() + this.right.eval();
        }
    },
    Mult {
        Exp left;
        Exp right;

        public int eval() {
            return this.left.eval() * this.right.eval();
        }
    };


    public abstract int eval();
}

enum ExpTwo{
    Int,
    Plus,
    Mult
}
```

Ansatz mit Klassen

```
public class Expression {
    public static void main(String[] args) {
        System.out.println("test");
    }
}

abstract class Exp{abstract public int eval();}
class IntExp extends Exp{
    public int val;
    public IntExp(int val){
        this.val = val;
    }
    @Override
    public int eval() {
        return val;
    }
}
class PlusExp extends Exp{
    public Exp left;
    public Exp right;
    public PlusExp(Exp left, Exp right){
        this.left = left;
```

```
        this.right = right;
    }
    @Override
    public int eval() {
        return left.eval() + right.eval();
    }
}

class MultExp extends Exp{
    public Exp left;
    public Exp right;

    public MultExp(Exp left, Exp right){
        this.left = left;
        this.right = right;
    }

    @Override
    public int eval() {
        return left.eval() * right.eval();
    }
}
```

2.4.3 Java Enums am Limit

- Idee, was aber wenn die Instanz ein Wrapper ist
- statische variablen schneiden uns

```
public class EnumLimit{
        public static void main(String[] args) {
                Animal a = Animal.Dog;
                Animal a2 = Animal.Dog;
                Animal b = Animal.Cat;
                System.out.println(a.getObject());
                System.out.println(a2.getObject());
                System.out.println(b.getObject());
                a.setObject("new Dog Value");
                b.setObject("new Cat value");
                System.out.println(a.getObject());
                System.out.println(a.getObject());
                System.out.println(b.getObject());
        }
}

enum Animal{
        Dog(new Wrapper("Doggy")),
        Cat(new Wrapper("Catty"));

        private Wrapper w;
        private Animal(Wrapper w){
```

```
                this.w = w;
        }

        public Object getObject(){
                return w.item;
        }
        public void setObject(Object o){
                w.item = o;
        }
}

class Wrapper{
        Object item;

        public Wrapper(Object o){
                item = o;
        }
}
```

output

```
Doggy
Doggy
Catty
new Dog Value
new Dog Value
new Cat value
```

3 Traits

3.1 Traits in Rust

1. geteilte funktionalität mit anderen Typen
2. Funktionsmenge über einem Typen
3. Oft mit Interfaces verglichen, sind aber keine Interfaces
4. interfaces sind Typen
5. adressieren ähnliche Probleme, traits aber mächtiger

3.1.1 Einfacher Trait

1. Prädikat auf einem Typen

```
trait Shape{
    fn area(s: &Self) ->i32;
}

struct Square{
    a: i32
```

```
}

impl Shape for Square {
    fn area(s: &Self)->i32{
        s.a*s.a
    }
}

struct Rectangle{
    a: i32,
    b: i32
}

impl Shape for Rectangle {
    fn area(s: &Self)->i32{
        s.a*s.b
    }
}
```

3.1.2 Shorthand Schreibweise

Andere Schreibweise, so kann man die Funktion auf einer Instanz des Structs aufrufen

```
trait Shape{
   fn area(&self) -> String;
}

impl Shape for Square{
   fn area(&self) -> i32{
       self.a*self.a
   }
}

fn main() {
   let s = Square{a: 10};
   print!("{}", s.area());
}
```

3.1.3 Default-Implementationen

1. geht in java auch

```
fn main() {
    let c1:Cat  = Cat{};
    Animal::makeNoise(&c1);
}

trait Animal{
    fn makeNoise(s: &Self){
```

```
        println!("The Animal made a noise");
    }
}
struct Cat{}
impl Animal for Cat{}
```

When running main yields

```
The Animal made a noise
```

3.1.4 Trait Bounds

```
//Das Shape Prädikat muss für A und für B gelten
fn sum_area<A:Shape,B:Shape>(x : &A, y : &B) -> i32 {
    return area(x) + area(y)
 }
```

3.1.5 Multiples Binding

Man kann auch Prädikate/Traits verunden

```
fn sum_area<A:Shape+OtherTraits>(x : &+OtherTraits) -> i32 {
        ...
 }
```

3.1.6 Dynamische Traits

Repräsentieren von Interfaces in Rust
Können Konkrete Typen als Parameter und Rückgabewerte nutzen

```
fn sum_area(x : Box<dyn Shape>, y: Box<dyn Shape>) -> i32 {
    return area(x) + area(y)
 }
```

3.1.7 Kurzschreibweise für dynamische Traits

```
fn sum_area(x : &(impl Shape), y: &(impl Shape)) -> i32 {
    return area(x) + area(y)
 }
```

3.1.8 Platzhaltertypen

```
fn main(){
    let m = Machine{};
    let a: i8 = 16;
    let b: i32 = TransformAB::transform(&m, a);
}
trait TransformAB{
    type A;
    type B;
    fn transform(s: &Self, a: Self::A) -> Self::B;
}
```

```rust
struct Machine{}
impl TransformAB for Machine{
    type A = i8;
    type B = i32;
    fn transform(s: &Self, a: Self::A) -> Self::B {
        i32::from(a)
    }
}
```

3.1.9 Assoziierte Konstanten

```rust
fn main(){
    let m = Machine{};
    let a: i8 = 16;
    let b: Vec<i32> = TransformAB::transform(&m, a);
}
trait TransformAB{
    type A;
    type B;
    const TIMES: u8;
    fn transform(s: &Self, a: Self::A) -> Vec<Self::B>;
}

struct Machine{}
impl TransformAB for Machine{
    type A = i8;
    type B = i32;
    const TIMES:u8 = 50;
    fn transform(s: &Self, a: Self::A) -> Vec<Self::B>{
        let mut v = Vec::new();
        let a32 = i32::from(a);
        for i in 0..Self::TIMES  {
            v.push(a32);
        }
        v
    }
}
```

3.1.10 Supertraits

- man kann hirarchie nachbauen

```rust
fn main() {
    let s = HskaStudent{name:"Mario", university:"hska", fav_language:"rust", git_username
    comp_sci_student_greeting(&s);
}
trait Person {
    fn name(&self) -> String;
}
trait Student: Person {
```

```
    fn university(&self) -> String;
}
trait Programmer {
    fn fav_language(&self) -> String;
}
trait CompSciStudent: Programmer + Student {
    fn git_username(&self) -> String;
}
fn comp_sci_student_greeting<S: CompSciStudent>(student: &S) {
    println!("Hey my name is {}, I study at {}. My favorite language is {} and my git user
}
struct HskaStudent{
    name: &'static str,
    university: &'static str,
    fav_language: &'static str,
    git_username: &'static str,
}
impl Person for HskaStudent{
    fn name(&self) -> String{
        self.name.to_string()
    }
}
impl Student for HskaStudent{
    fn university(&self) -> String {
        String::from(self.university)
    }
}

impl Programmer for HskaStudent{
    fn fav_language(&self) -> String{
        String::from(self.fav_language)
    }
}
impl CompSciStudent for HskaStudent{
    fn git_username(&self) -> String {
        String::from(self.git_username)
    }
}
```

3.2 Mächtigkeit von Traits

3.2.1 Gleiche Methodensignatur

- kurzschreibweise geht hier nicht

```
fn main() {
   let x = some_struct{};
   musicplayer::play(&x);
   boardgame::stop(&x);
}
```

```
struct some_struct{}

trait musicplayer{
    fn play(s: &Self);
    fn stop(&self);
}
trait boardgame{
    fn play(s: &Self);
    fn stop(&self);
}

impl musicplayer for some_struct {
  fn play(s: &Self) {
      println!("Playing music");
  }
  fn stop(&self) {
      println!("Stopping music");
  }
}
impl boardgame for some_struct {
  fn play(s: &Self) {
      println!("Playing boardgame");
  }
  fn stop(&self) {
      println!("Stopping boardgame");
  }
}
```

3.2.2 Generische Mehrfachimplementierung

```
fn main() {
    let s : some_struct = some_struct{};
    let someInteger: i32 = s.mygenval();
    let someString: String = s.mygenval();
}
struct some_struct{}

trait generic<T>{
   fn mygenval(&self) -> T;
}

impl generic<i32> for some_struct {
   fn mygenval(&self) -> i32{
       5
   }
}

impl generic<String> for some_struct {
   fn mygenval(&self) -> String{
       "abc".to_string()
```

```
   }
}
```

3.2.3 Referenzierung des eigenen Typen

```
trait genCopy{
    fn genCopy(s: &Self) -> Self;
}

struct Dog{
    name: String,
    age: u8,
}
struct Cat{
    name: String,
    age: u8,
}

impl genCopy for Dog{
    fn genCopy(s: &Self) -> Self {
        return Dog{name: s.name.clone(), age: s.age};
    }
}
impl genCopy for Cat{
    fn genCopy(s: &Self) -> Self {
        return Cat{name: s.name.clone(), age: s.age};
    }
}
```

3.2.4 Funktionalität für Third-Party-Datentypen

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

fn main() {
    thiryparty_struct{}.sleep();
}
struct thiryparty_struct{}

trait Sleep{
    fn sleep(&self);
}

impl Sleep for thiryparty_struct {
    fn sleep(&self){
        thread::sleep(Duration::from_millis(1000));
    }
}
```

3.2.5 Referenzierung des eigenen Typen

3.2.6 Konditionelle Implementierung

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

struct dog{
    name: String,
    age: u8,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

3.3 Traitfunktionalität in Java

3.3.1 Gleiche Methodensignatur

Geht nicht, weil nicht eindeutig (Signaturkonflikt)

```
class SomeClass implements musicplayer, boardgame{
   public void play(){
       System.out.println("You are playing");
   }
}
interface musicplayer{
   public void play();
}
interface boardgame{
   public void play();
}
```

Lösung via Adapterpattern

```
public class AdapterCompatible {
    public static void main(String[] args) {
```

```
        SomeClass sc = new SomeClass();
        MusicPlayerAdapter ma = new MusicPlayerAdapter(sc);
        BoardGameAdapter ba = new BoardGameAdapter(sc);
        ma.play();
        ba.play();
    }
}
class SomeClass{
    public void playMusic(){
        System.out.println("Playing music");
    }
    public void playBoardGame(){
        System.out.println("Playing boardgame");
    }
}
interface MusicPlayer{
    public void play();
}
interface BoardGame{
    public void play();
}

class MusicPlayerAdapter implements MusicPlayer {
    private SomeClass someClass;

    public MusicPlayerAdapter(SomeClass someClass) {
        this.someClass = someClass;
    }
    @Override
    public void play() {
        someClass.playMusic();
    }
}

class BoardGameAdapter implements BoardGame {
    private SomeClass someClass;

    public BoardGameAdapter(SomeClass someClass) {
        this.someClass = someClass;
    }
    @Override
    public void play() {
        someClass.playBoardGame();
    }
}
```

3.3.2 Generische Mehrfachimplementierung

- Interface kann nicht mehr als einmal implementiert werden - Wieder Adapter

```
public class SomeClass implements Generic<Integer>, Generic<String> {
    public static void main(String[] args) {
        SomeClass sc = new SomeClass();
    }
}

interface Generic<T> {
    public T mygenvalue();
}
```

output

```
SomeClass.java:1: error: Generic cannot be inherited with
different arguments: <java.lang.Integer> and <java.lang.String>
```

Lösung

```
public class SomeClass {
    public static void main(String[] args) {
        SomeClass sc = new SomeClass();
        Integer someInt = new GenericIntAdapter(sc).mygenvalue();
        String someString = new GenericStringAdapter(sc).mygenvalue();
    }
}

interface Generic<T> {
    public T mygenvalue();
}

class GenericIntAdapter implements Generic<Integer> {
    private SomeClass someClass;

    public GenericIntAdapter(SomeClass someClass) {
        this.someClass = someClass;
    }

    @Override
    public Integer mygenvalue() {
        return 5;
    }
}
class GenericStringAdapter implements Generic<String> {
    private SomeClass someClass;

    public GenericStringAdapter(SomeClass someClass){
        this.someClass = someClass;
    }
    @Override
    public String mygenvalue() {
        return "abc";
```

```
    }
}
```

3.3.3 Funktionalität für Third-Party-Datentypen

```
public class ThirdParty {
    public static void main(String[] args) {
        ThirdParty original = new ThirdParty();
        WrapperClass wrapper = new WrapperClass(original);

        wrapper.doSomething();
        wrapper.sleep();
        wrapper.doSomething();
    }
    public void doSomething() {
        System.out.println("Doing something...");
    }
}

class WrapperClass {
    private ThirdParty original;

    public WrapperClass(ThirdParty original) {
        this.original = original;
    }

    public void doSomething() {
        original.doSomething();
    }

    public void sleep() {
        try {
            Thread.sleep(1000); // Sleep for 1000 milliseconds
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

3.3.4 Referenzierung des eigenen Typens

```
public class SameInputOutput{
    public static void main(String[] args) {

    }
}
interface sameObject<T>{
    public T returnSameObject(T input);
}
```

```
class Dog implements sameObject<Dog>{
    public Dog returnSameObject(Dog input){
        return input;
    }
}
```

3.3.5 Konditionelle Implementierung

Ein bisschen anders, da das Pair jetzt nur werte zulässt die von der Abstrakten Klasse comparable erben

```
public class Conditional {
    public static void main(String[] args) {
        Pair<Integer> intpair = new Pair(1,2);
        Pair<Dog> dogpair = new Pair(new Dog("Ben"), new Dog("Albert"));
    }
}

class Pair<T extends Comparable<T>>{
    private T x;
    private T y;

    public Pair(T x, T y) {
        this.x = x;
        this.y = y;
    }

    public void cmpDisplay() {
        if (x.compareTo(y) >= 0) {
            System.out.println("The largest member is x = " + x);
        } else {
            System.out.println("The largest member is y = " + y);
        }
    }
}

class Dog{
    private String name;
    public Dog(String name){
        this.name = name;
    }
}
```

compiler output

```
Conditional.java:4: error: type argument Dog is not within bounds of type-variable T
Pair<Dog> dogpair = new Pair(new Dog("Ben"), new Dog("Albert"));
          ^
where T is a type-variable:
T extends Comparable<T> declared in class Pair
Conditional.java:4: error: incompatible types: Dog cannot be converted to Comparable
Pair<Dog> dogpair = new Pair(new Dog("Ben"), new Dog("Albert"));
```

Eine Idee wäre noch dynamisches Checken mit instance of, das ist aber Fehleranfällig.

4 Vergleich

4.1 Enums

- Enums kombiniert mit Klassen kann Alle Enums nachbauen

4.2 Traits

Literatur

[1] Slides about haskell by prof. dr. sulzmann. `https://sulzmann.github.io/ProgrammingParadigms/lec-rust-vs-haskell.html`.

[2] The rust programming language. `https://doc.rust-lang.org/stable/book/`.

[3] Slides about rust by prof. dr. sulzmann. `https://sulzmann.github.io/ProgrammingParadigms/lec-rust.html`.

[4] Design patterns. `https://refactoring.guru/design-patterns`.