

# Seminararbeit Traits und Enums in Rust

Mario Occhinegro  
HKA University of Applied Sciences

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Enums</b>	<b>1</b>
2.1	Enums in Rust . . . . .	1
2.1.1	Normale Enums . . . . .	1
2.1.2	Enum mit Werten . . . . .	1
2.1.3	Enum mit Funktionen . . . . .	2
2.2	Enums in Java . . . . .	2
2.2.1	Normale Enums . . . . .	2
2.2.2	Enums mit Werten . . . . .	2
2.2.3	Enum mit Funktionen . . . . .	3
2.3	Mächtigkeit von Rust Enums . . . . .	3
2.3.1	Der Enum als algebraischer Datentyp . . . . .	3
2.3.2	Generische Enums . . . . .	4
2.3.3	Rust Enums und die Vermeidung von Nullpointer-Ausnahmen . . . . .	4
2.3.4	Rekursive Enums . . . . .	4
2.3.5	Match Statement . . . . .	5
2.3.6	Feste Enum Cases . . . . .	5
2.3.7	Nested Pattern Matching . . . . .	7
2.3.8	Erweiterbare Funktionen für Enums . . . . .	8
2.4	Rust-Enum-Funktionalität in Java . . . . .	8
2.4.1	Expression-Logik in Java . . . . .	8
2.4.2	Java Enums am Limit - Wrapperinstanz für den Typ . . . . .	10
2.4.3	Enums kombiniert mit Klassen . . . . .	11
<b>3</b>	<b>Traits</b>	<b>11</b>
3.1	Traits in Rust . . . . .	11
3.1.1	Traits sind keine Typen . . . . .	11
3.1.2	Default-Implementationen . . . . .	12
3.1.3	Trait Bounds . . . . .	12
3.1.4	Multiples Binding . . . . .	12
3.1.5	Dynamische Traits . . . . .	12
3.1.6	Kurzschreibweise für dynamische Traits . . . . .	13
3.1.7	Shorthand Schreibweise . . . . .	13
3.1.8	Schreibweise bei Uneindeutigkeit . . . . .	13
3.1.9	Supertraits . . . . .	14
3.1.10	Referenzierung des eigenen Typen . . . . .	15
3.1.11	Platzhaltertypen . . . . .	16
3.1.12	Assoziierte Konstanten . . . . .	16
3.2	Interfaces in Java . . . . .	17
3.3	Mächtigkeit von Traits . . . . .	17
3.3.1	Nicht vereinbare Interfaces . . . . .	17
3.3.2	Generische Mehrfachimplementierung . . . . .	17
3.3.3	Funktionalität für Third-Party-Datentypen . . . . .	17
3.3.4	Konditionelle Implementierung . . . . .	17
3.4	Traitfunktionalität in Java . . . . .	18
3.4.1	Nicht vereinbare Interfaces (Java) . . . . .	18
3.4.2	Generische Mehrfachimplementierung (Java) . . . . .	18

3.4.3	Funktionalität für Third-Party-Datentypen (Java) . . . .	18
3.4.4	Konditionelle Implementierung (Java) . . . . .	18
<b>4</b>	<b>Vergleich</b>	<b>18</b>

# 1 Einleitung

## 2 Enums

### 2.1 Enums in Rust

#### 2.1.1 Normale Enums

Enumerationstypen

Auf den ersten Blick identisch.

Java Enum:

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

#### 2.1.2 Enum mit Werten

```
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

```
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

```

    Compiling playground v0.1.0 (/home/mario/Desktop/seminararbeit/rust/playground)
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 wildcard
   |
30 | ~         Exp::Mult{left, right} => left.eval() * right.eval(),
31 | +         &Exp::Div { .. } => todo!()
   |

```

For more information about this error, try `rustc --explain E0004`.  
error: could not compile `playground` due to previous error

### 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

```

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()
        }
    }
    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 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();
    }
}
```

#### 2.4.2 Java Enums am Limit - Wrapperinstanz für den Typ

- Idee, was aber wenn die Instanz ein Wrapper ist

- nicht sehr ergonomisch
- statische variablen schneiden uns

```

mintedpublic class playground{
    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
```

#### 2.4.3 Enums kombiniert mit Klassen

## 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 Traits sind keine Typen

1. Prädikat auf einem Typen

```
mintedtrait 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 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

### 3.1.3 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)
}
```

### 3.1.4 Multiples Binding

Man kann auch Prädikate/Traits verunden

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

### 3.1.5 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)
}
```

### 3.1.6 Kurzschreibweise für dynamische Traits

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



### 3.1.7 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());
}
```

### 3.1.8 Schreibweise bei Uneindeutigkeit

```
fn main() {
    let s = Square{a: 10};
    print!("{}", Shape2::area(&s));
}

struct Square{
    a: u32,
}
trait Shape{
    fn area(&self) -> u32;
}

impl Shape for Square{
    fn area(&self) -> u32{
        self.a*self.a
    }
}

trait Shape2{
    fn area(&self) -> u32;
}

impl Shape2 for Square{
    fn area(&self) -> u32{
        self.a*self.a
    }
}

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

```

}

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

trait Shape2{
    fn area(&self) -> String;
}

impl Shape2 for Square{
    fn area(&self) -> i32{
        self.a*self.a
    }
}

fn main() {
    let s = Square{a: 10};
    print!("{}", Shape2::area(s));
}

```

### 3.1.9 Supertraits

- man kann hierarchie nachbauen

```

fn main() {
    let s = HskaStudent{name:"Mario", university:"hska", fav_language:"rust", git_username:"mario", comp_sci_student_greeting:&greeting};
    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 is {}", student.name(), student.university(), student.fav_language(), student.git_username());
}

struct HskaStudent{
    name: &'static str,
    university: &'static str,
    fav_language: &'static str,
    git_username: &'static str,
    comp_sci_student_greeting: &'static fn(&S) where S: CompSciStudent,
}

```

```

        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)
        }
    }
}

```

### 3.1.10 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};
    }
}

```

### 3.1.11 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;
}

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

### 3.1.12 Assoziierte Konstanten

```
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
    }
}
```

## 3.2 Interfaces in Java

Wird oft hiermit verglichen, ist aber anders. Interfaces sind Typen

## 3.3 Mächtigkeit von Traits

### 3.3.1 Nicht vereinbare Interfaces

### 3.3.2 Generische Mehrfachimplementierung

### 3.3.3 Funktionalität für Third-Party-Datentypen

### 3.3.4 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);
        }
    }
}
```

## 3.4 Traitfunktionalität in Java

### 3.4.1 Nicht vereinbare Interfaces (Java)

### 3.4.2 Generische Mehrfachimplementierung (Java)

### 3.4.3 Funktionalität für Third-Party-Datentypen (Java)

### 3.4.4 Konditionelle Implementierung (Java)

## 4 Vergleich