Traits und Enums in Rust

Enums im Vergleich

- Java Enums
  - Finale Instanzen
  - Variablen Statisch
  - Switch case nicht fix
- Rust Enums
  - algebraische Datentypen
  - Werte des zugehörigen Typs sind veränderbar
  - Match Cases über Enum sind fix

Enums in Java

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

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

   public final String label;
   public final int weight;

   //constructor
   private AnimalWithValues(String label, int weight){
       this.label= label;
       this.weight = weight;
   }
   //Instanzmethode -> compiler macht Instanzen von unseren Enum-Typen
   public boolean isCat(){
       if (this == AnimalWithValues.Cat){
           return true;
       }else{
           return false;
       }
   }
}
```

Enums in Rust

```
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,
       }
   }

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

Vorteile von Rust Enums

```
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()

        }
    }
}
```

Kann Java das auch? Versuch 1

```
enum Exp {
   Int {
       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();
}
```

Kann Java das auch? Versuch 1

```
enum Exp {
   Int {
       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();
}
```

Kann Java das auch? Versuch 2

```
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();
   }
}
```

Interessantes zu Enums

```
fn 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
    }
}
```

Interessantes zu Enums

```rust
pub enum Option<T> {
    /// No value.
    #[lang = "None"]
    #[stable(feature = "rust1", since = "1.0.0")]
    None,
    /// Some value of type `T`.
    #[lang = "Some"]
    #[stable(feature = "rust1", since = "1.0.0")]
    Some(#[stable(feature = "rust1", since = "1.0.0")] T),
}
```

Nested Patterns

```
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()
           }

      }
  }
}
```

“Nested Patterns” in Java

```
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() {
       int leftValue = left.eval();
       if(left instanceof IntExp && leftValue == 0){
           return 0;
       }else {
           return left.eval() * right.eval();
       }
   }
}
```

Interfaces und Traits

Interfaces

- Interfaces sind Typen
- Wie ein Vertrag

Traits

- Mengen von Funktionen (über einem Typ z.B. area: rectangle -> N)
- Prädikat
- Erlauben Einschränkung

Interfaces in Java

```
interface Shape{
   public int area();
}
class Square implements Shape{
   public int x;
   @Override
   public int area() {
       return x*x;
   }
}

class Rectangle implements Shape{
   public int x;
   public int y;
   @Override
   public int area() {
       return x*y;
   }
}
```

Traits in Rust

```
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
   }
}

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

Shorthand Schreibweise

```
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())
}
```

Szenario: Nicht Vereinbare Interfaces (z.B. 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();
}
```

Szenario: Nicht Vereinbare Interfaces (z.B. 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();
}
```

Szenario: Nicht Vereinbare Interfaces (Rust)

```rust
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);
}
```

```rust
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");
  }
}
```

Szenario: Nicht Vereinbare Interfaces (Adapter-Lösung)

```
class SomeClass{
    public void playMusic(){
        System.out.println("Playing music");
    }
    public void playBoardGame(){
        System.out.println("Playing music");
    }
}

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();
    }
}
```

Szenario: Funktionalität für jeden Datentyp

- Wir können einfach für ThirdPartyStructs Funktionalität/Schnittstellen implementieren

- Java kann hier wieder auf Design-Patterns (Wrapper-Pattern) zurückgreifen.

Szenario: Konditionelle Implementierungen

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

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);
        }
    }
}
```

```
fn main(){
    let np = Pair{x: 3, y:4};
    let sp = Pair{x: "abc".to_string(),y:"def".to_string()};
    let d1 = dog{name: "Robert".to_string(), age: 7};
    let d2 = dog{name: "Paul".to_string(), age: 7};
    let dp = Pair{x:d1, y:d2};

    np.cmp_display();
    sp.cmp_display();
    //dp.cmp_display();
}

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

Szenario: Konditionelle Implementierungen (Java Version)

```java
public class ConditionalExample
   public static void main(String[] args) {
       //Accepted
       Pair<NoHello> normalPair = new Pair(new NoHello(), new NoHello());
       //Not accepted
       SpecificPair<NoHello> deniedspecificPair =
       new SpecificPair(new NoHello(), new NoHello());
       //Accpeted
       SpecificPair<Hello> acceptedSpecificPair =
       new SpecificPair(new Hello(), new Hello());
   }
}

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

class SpecificPair<T extends IGiveHelloString> extends Pair{
   public SpecificPair(T x, T y){
       super(x,y);
   }

   public void PrintBothHelloString(){
   }
}
```

```java
class NoHello{
   public NoHello(){}
}
class Hello implements IGiveHelloString{

   @Override
   public String sayHello() {
       return "Hello";
   }



}

interface IGiveHelloString{
   public String sayHello();
}
```

Vergleich Java Enums vs Rust Enums

Java

- Komplexe Datenstrukturen nur über Klassen
- Klasse selbst bei erweitert werden um Funktionalität zu bieten
- Methoden für Klassen nicht entkoppelt erweiterbar
- Switch Case nicht abgeschlossen und nur simple Typen
- Feste Modelierung für alle Optionen

Rust

- Abgeschlossene Fälle beim Pattern Matching
- Funktionen für Enums sind entkoppelt erweiterbar
- Beliebige Modelierung der Optionen

Vergleich Java Interfaces, Classes vs Rust Traits

Java

- Interfaces sind Typen
- Klasse selbst müssen verändert werden um Implementation zu gewährleisten
- Erweitern der Funktionalität ebenfalls nur durch Änderung in der Klasse selbst

Rust

- Traits sind Funktionsmengen über Typ
- Dynamische Traits sind wie Interfaces
- Prädikate
- Implementationen von Traits entkoppelt von Datentyp

Danke für eure Aufmerksamkeit

noch Fragen?