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
        }
    }
```

Enums in Rust

```
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()

        }
    }
}
```

Vorteile von Rust Enums und Pattern Matching

- Kompakt
- Sehr performant
- Stärkere Typsicherheit => weniger Laufzeitfehler
- Beziehungen zwischen Varianten einfach nachvollziehbar
- Abgeschlossenes Pattern Matching
- Wiederverwendbarkeit durch Generische Enums
- Nested Pattern Matching

Kann Java das auch?

```
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()
            }

      }
  }
}
```

Interfaces und Traits

Interfaces

- Interfaces sind typisieren Klassen
- Definiere Verträge. Implementiert Klasse Interface=> muss Vertrag einhalten
- Klasse kann mehrere Interfaces Implementieren

Traits

- Mengen von Funktionen über Typ
- Prädikat, das gilt, wenn Implementation für Typ existiert
- Erlauben Einschränkung von Parametern und Rückgabewerten

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: Konditionelle Implementierungen

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

Vergleich Rust Enums vs Java - Rust

Vorteile

- Laufzeit Typsicherheit durch PM
- Leistungsoptimierung (Checks bei Kompilierzeit)
- Kompakt
- Leicht lesbar
- Zustände und Verhalten immer klar definiert

Nachteile

- Verändern Varianten
  => Codeanpassung überall wo PM verwendet wird
- Komplexe Enum Struktur
  => Unübersichtliches Pattern Matching
  => Schwer erweiterbar

Vergleich Rust Enums vs Java - Java

Vorteile

- Leicht erweiterbar (Bestehender Code muss nicht angepasst werden, wenn neue Klasse dazukommt)
- Gemeinsame Schnittstellen => Code flexibler
- Leicht erweiterbar mit Klassenhirarchie / Interfaces
- Funktionalität an Klasse gekoppelt => übersichtlicher

Nachteile

- Mehr Boilerplate-Code
- Mögliche Typunsicherheiten

Vergleich Java Interfaces, Classes vs Rust Traits - Rust

Vorteile

- Beliebige funktionelle Erweiterung von Structs
- Erweiterung von 3rd Party Structs
- Konditionelle Implementierung mit Trait Bounds möglich
- Weniger Probleme mit Interface-Konflikten

Nachteile

- Keine statischen Variablen zwischen Implementationen
- Implementierung eines Supertraits nicht mit Untertrait teilbar
  => Implementation nicht wiederverwendbar

Vergleich Java Interfaces, Classes vs Rust Traits - Java

Vorteile

- Vererbung von Datenfeldern und Funktionen möglich
- Überschreibung von Methoden der Übergeordneten Klasse möglich
- Funktionalität in Unterklasse wiederverwendbar
  => weniger Code Redundanz

Nachteile

- Manche Funktionalitäten nur über Umwege erreichbar (z.B. Designpatterns)
- Funktionelle Erweiterung nur durch Veränderung der Klasse selbst möglich
- Viel Boilerplate

Danke für eure Aufmerksamkeit

noch Fragen?