Seminararbeit Traits und Enums in Rust

Mario Occhinegro
HKA University of Applied Sciences

Inhaltsverzeichnis

Zusammenfassung

1 Einleitung

2 Enums

Enumerationstypen
Auf den ersten Blick identisch.
Java Enum:

```
enum Color{
    red,
    green,
    blue;
}
```

Rust Enum:

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

Auf Konkretere Unterschiede gehen wir jetzt ein

2.1 Enums in Rust

- Algebraische Datentypen

2.1.1 Match Statement

- abgeschlossen
- an Haskell angelehnt
- mehr als nur if else
- sehr ergonomisch, aussagekräftig und kurz

```
fn main() {
        let a1 = Animal::Dog;
        match a1{
                Animal::Dog => println!("It's a Dog"),
                Animal::Cat => println!("It's a Cat"),
                Animal::Bird => println!("It's a Bird")
        }
}

enum Animal{
```

```
        Dog,
        Cat,
        Bird
}
```

2.1.2 Der Enum als algebraischer Datentyp

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

```
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,
        }
    }
}
```

2.1.3 Rekusive Enums und Datentypen

- braucht Box (wie Zeiger)
- Box sonst, rekursive Definition ohne Direktion

```
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()

        }
    }
}
```

2.1.4 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()
            }
       }
   }
}
```

2.1.5 Generische Enums

- Enums können mit generischen Werten generiert werden

```
enum Option<T> {
        None,
```

```
        Some(T),
}
```

2.1.6 Verwendung des Rust Enums zur 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
    }
}
```

2.2 Enums in Java

- Enums 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 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;
    }
}
```

2.2.3 Enum Funktionen

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

2.3 Vergleich von Java und Rust Enums

2.4 Rust Enum Implementationsbeispiele

2.4.1

2.5 Beispielfunktionalität in Java

2.5.1 Expression-Logik in Java

Naiver Ansatz

```
mintedpublic 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();}
```

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

2.5.2 Java Enums am Limit - Idee einer 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

```
minted
```

3 Traits

3.1 Allgemeines zu Traits

3.1.1 Traits sind keine Typen

3.2 Traits in Rust

3.2.1 Default-Implementationen

3.2.2 Trait Bounds

Mehrfaches Traitbinding

Konditionelle Implementierung mit Trait Bounds

3.2.3 Dynamische Traits

3.2.4 Traits als Parameter

wie interfaces

3.2.5 Supertraits

3.2.6 Referenzierung des eigenen Typen

3.2.7 Spezifizierung von Platzhaltertypen

3.2.8 Shorthand Schreibweise

3.2.9 Schreibeweise bei Uneindeutigkeit

3.3 Rust Trait Beispiele

3.4 Beispielfunktionalität in Java

4 Vergleich der beiden Ansätze