

Traits und Enums in Rust

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

```
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),  
}
```

Interfaces und Traits

Interfaces

- Typen
- Wie ein Vertrag

Traits

- Mengen von Funktionen (über einem Typ z.B. $\text{area: rectangle} \rightarrow \mathbb{N}$)
- Prädikat
- Einschränkung auf Typ

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

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

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

Interface Inheritance

```
interface Person{  
    public String name();  
}
```

```
interface Student extends Person{  
    public String university();  
}
```

Supertraits

```
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) {  
    ...  
}
```

Szenario: Nicht Vereinbare Interfaces

```
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

```
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)

```
fn main() {  
    let x = some_struct{x:10};  
    A::hello(&x);  
    B::hello(&x);  
    A::goodbye(&x);  
    B::goodbye(&x);  
  
}  
  
struct some_struct{  
    x: i32,  
}  
  
trait A{  
    fn hello(s: &Self);  
    fn goodbye(&self);  
}  
  
trait B{  
    fn hello(s: &Self);  
    fn goodbye(&self);  
}  
  
impl A for some_struct {  
    fn hello(s: &Self) {  
        println!("Hello from A");  
    }  
    fn goodbye(&self) {  
        println!("Goodbye from A");  
    }  
}  
  
impl B for some_struct {  
    fn hello(s: &Self) {  
        println!("Hello from B");  
    }  
    fn goodbye(&self) {  
        println!("Goodbye from B");  
    }  
}
```

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: Generische Mehrfachimplementierung (Java)

```
interface MyGenericInterface<T> {  
    T getValue();  
    void setValue(T value);  
}  
  
public class MultipleImplementations implements MyGenericInterface<Integer>, MyGenericInterface<String>{  
    String val1 = "abc";  
    int val2 = 0;  
  
    @Override  
    public Integer getValue() {  
        return val2;  
    }  
  
    @Override  
    public void setValue(Integer value) {  
        this.val2 = value;  
    }  
  
    @Override  
    public String getValue() {  
        return val1;  
    }  
  
    @Override  
    public void setValue(String value) {  
        this.val1 = value;  
    }  
}
```

Szenario: Generische Mehrfachimplementierung (Java)

```
interface MyGenericInterface<T> {  
    T getValue();  
    void setValue(T value);  
}  
  
public class MultipleImplementations implements MyGenericInterface<Integer>, MyGenericInterface<String>{  
    String val1 = "abc";  
    int val2 = 0;  
  
    @Override  
    public Integer getValue() {  
        return val2;  
    }  
  
    @Override  
    public void setValue(Integer value) {  
        this.val2 = value;  
    }  
  
    @Override  
    public String getValue() {  
        return val1;  
    }  
  
    @Override  
    public void setValue(String value) {  
        this.val1 = value;  
    }  
}
```



Szenario: Generische Mehrfachimplementierung (Rust)

```
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()
    }
}
```

Szenario: Generische Mehrfachimplementierung (Adapter)

```
class MultipleImplementations {
    String val1 = "abc";
    int val2 = 0;
}

class IntegerAdapter implements MyGenericInterface<Integer> {
    private MultipleImplementations mi;

    public IntegerAdapter(MultipleImplementations instance) {
        this.mi = instance;
    }

    @Override
    public Integer getValue() {
        return mi.val2;
    }

    @Override
    public void setValue(Integer value) {
        mi.val2 = value;
    }
}
```

```
class StringAdapter implements MyGenericInterface<String> {
    private MultipleImplementations mi;

    public StringAdapter(MultipleImplementations instance) {
        this.mi = instance;
    }

    @Override
    public String getValue() {
        return mi.val1;
    }

    @Override
    public void setValue(String value) {
        mi.val1 = value;
    }
}
```

Szenario: Funktionalität für jeden Datentyp

- Wir können einfach für ThirdPartyStructs Funktionalität/Schnittstellen implementieren
- Java kann hier wieder auf einen Adapter 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)

- geht nicht leider nicht
- begrenzte konditionelle implementierung auf Typ möglich, aber nicht das gleiche z.B.

```
public void doSomething(Object value) {  
    if (value instanceof Integer) {  
        doIntegerThing((Integer) value);  
    } else if (value instanceof String) {  
        doStringThing((String) value);  
    } else {  
        throw new IllegalArgumentException("Unsupported type");  
    }  
}
```

Szenario: Szenario Input und Output 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};
  }
}
```

Szenario: Szenario Input und Output des eigenen Typen (Java)

- keine perfekte Lösung, geht aber nicht anders

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

Danke für eure Aufmerksamkeit

noch Fragen?

