Seminararbeit Traits und Enums in Rust

Mario Occhinegro
HKA University of Applied Sciences

Inhaltsverzeichnis

Zusammenfassung

Traits und Enums in Rust sind Programmierkonzepte, die bereits aus Haskell unter Namen wie Typklassen oder ADTs bekannt sind[1]. Die Konzepte haben einige Vorteile gegenüber normalen Enums in Java sowie Interfaces und Vererbung, da diese ähnliche Probleme zu lösen versuchen. Manche Funktionalitäten lassen sich mittels Designpatterns, wie dem Adapter-Pattern oder dem Wrapper-Pattern, lösen, wobei diese Lösungen meist unnötig verbos erscheinen. Gewisse Funktionalitäten können Sprachen wie Java aber auch gar nicht bereitstellen. Bezogen auf diese Konzepte ist Rust mit Java verglichen also syntaktisch prägnanter, lesbarer und stellenweise funktionell sogar überlegen.

1 Einleitung

Die Programmiersprachen Rust und Java bieten Entwicklern eine Vielzahl von Programmierkonzepten, um robusten und flexiblen Code zu schreiben. In diesem Vergleich konzentrieren wir uns auf zwei wichtige Konzepte: Enums und Traits. Da Java nicht mit Traits arbeitet, werden stattdessen die zur Verfügung stehenden Möglichkeiten mit Interfaces und Klassen-Hirarchie verglichen werden, da diese ähnliche Probleme wie Traits adressieren.
In Rust sind Enums algebraische Datentypen, die ursprünglich aus der funktionalen Programmiersprache Haskell stammen. Sie ermöglichen es, eine feste Anzahl von Varianten zu definieren, die jeweils spezifische Informationen enthalten können. Mit Pattern Matching ist es in Rust möglich, auf die verschiedenen Varianten eines Enums zuzugreifen und sie zu verarbeiten. Ein bemerkenswertes Merkmal des Rust Enum Pattern Matching ist, dass es abgeschlossen ist, das heißt, jeder Fall des Enums muss abgedeckt werden. Dadurch wird vermieden, dass potenzielle Fehler unbemerkt bleiben, und der Code wird robuster. Ein weiteres leistungsstarkes Konzept in Rust sind Traits. Diese können als Mengen über einem Typ betrachtet werden und verhalten sich Typklassen in Haskell. Ähnlich zu Interfaces und Vererbung in Java ermöglichen Traits in Rust eine Möglichkeit, Funktionalität zwischen verschiedenen Datentypen zu teilen. Sie dienen als Prädikate für Typen[3] und erlauben es, bestimmte Verhaltensweisen festzulegen, ohne eine konkrete Implementierung vorzugeben. Dies ermöglicht uns, flexiblen und modularen Code zu schreiben. Im Vergleich zu Java zeigen Traits in Rust einige bemerkenswerte Vorteile. Während Java Interfaces zwar eine ähnliche Funktion bieten, aber oft mit Einschränkungen verbunden sind und Vererbung komplexe Klassenhierarchien erfordert, erlauben Traits in Rust eine granulare und unabhängige Zusammenarbeit von Typen. Sie bieten eine hohe Flexibilität bei der Kombination von Funktionen und vermeiden unnötige Abhängigkeiten zwischen Klassen. Später werden wir detaillierter auf diese Unterschiede eingehen und untersuchen, ob Java in der Lage ist, ähnliche Funktionalitäten wie Rust zu gewährleisten.

2 Enums

Enums sind eine Möglichkeit, eine begrenzte Anzahl von Optionen zu definieren. Sie dienen dazu, eine feste Menge von Werten darzustellen. Enums ermöglichen es, sichereren Code zu schreiben und diesen lesbarer zu gestalten.
Anhand einfacher Beispiele wird zunächst gezeigt, wie simple Enums, Enums mit zugehörigen Werten und Enums mit Funktionen in Rust und Java aussehen. Danach gehe ich auf die konkreten Unterschiede ein und zeige den Mehrwert, den Rust Enums bringen. Im Anschluss versuche ich die Funktionalität, die Rust Enums uns geben, in Java (falls es möglich ist) nachzubauen.

2.1 Enums in Rust

Auf den ersten Blick mögen Enums in Rust und Java ähnlich aussehen. Jedoch gibt es einen entscheidenden Unterschied: Rust Enums sind algebraische Datentypen. Algebraische Datentypen in Rust erlauben es, komplexe Datenstrukturen zu definieren, die weit über einfache Aufzählungen von Optionen hinausgehen. Sie können Varianten enthalten, die selbst wiederum Daten oder weitere Optionen beinhalten können. Diese Flexibilität bietet viele Möglichkeiten bei der Modellierung von Datentypen. Ein weiteres Merkmal von Rust Enums ist das abgeschlossene Pattern Matching. Dies bedeutet, dass bei der Verarbeitung eines Enums alle möglichen Fälle explizit abgedeckt werden müssen. Der Compiler erzwingt diese Vollständigkeit, was zu sichererem und fehlerfreiem Code führt. Dadurch wird vermieden, dass unbehandelte Fälle auftreten und potenziell zu Fehlern führen können. Im Gegensatz dazu bieten Java Enums keine eingebaute Unterstützung für algebraische Datentypen oder abgeschlossenes Pattern Matching. Dies kann dazu führen, dass beim Umgang mit Enums in Java unbeabsichtigte Fehler auftreten können, wenn nicht alle Varianten explizit behandelt werden. Durch die Kombination von algebraischen Datentypen und abgeschlossenem Pattern Matching bieten Rust Enums eine überlegene Möglichkeit, komplexe Datenstrukturen zu modellieren und zu verarbeiten. Diese Funktionalitäten machen Rust Enums zu einer mächtigen und robusten Technik, die über die Möglichkeiten von Java Enums hinausgeht. Im Bezug auf algebraische Datentypen muss Java also an dieser Stelle auf Modellierung durch Klassen zurückgreifen.

2.1.1 Normale Enums

Einfache Auflistungen sind in Rust und Java komplett identisch.

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

2.1.2 Enums mit Werten

Wenn wir jedem Enum einen konkreten Wert eines Typs zuordnen möchten, müssen wir das in Rust über eine Funktion machen. Das Pattern Matching, auf das ich nachher noch genauer eingehen werde, lässt uns an dieser Stelle bequem und übersichtlich die zugehörigen Werte festlegen. Bei Java ist es möglich, diese einer Option des Enums direkt anzugeben. Dies ist aber kein Nachteil, da es uns erlaubt, bei Bedarf weitere Werte hinzuzufügen, ohne das Enum zu verändern.

```
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 Enums mit Funktionen

Das Definieren von Funktionen über einem Enumtypen funktioniert analog wie das Hinzufügen von Werten.

```
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

In Java werden Enums als spezielle Klassen behandelt. Enumtypen in Java werden als Instanzen der jeweiligen Enumklasse behandelt. Jede Instanz repräsentiert dabei einen bestimmten Zustand oder eine Option. Dies ermöglicht eine einfache und intuitive Verwendung von Enums in Java. Eine wichtige Eigenschaft von Java Enums ist, dass ihre Instanzvariablen statisch sind. Dies bedeutet, dass die Werte der Instanzvariablen für jede einzelne Instanz des Enums gleich sind. Durch die Verwendung von speziellen Klassen und Instanzen ermöglichen Java Enums eine effiziente und sichere Handhabung einer begrenzten Menge von Optionen. Im Vergleich zu Rust Enums fehlen jedoch in Java einige der fortgeschritteneren Konzepte wie algebraische Datentypen und abgeschlossenes Pattern Matching. Dies kann dazu führen, dass die Handhabung und Verarbeitung von Enums in Java etwas umständlicher ist und es möglicherweise erforderlich ist, zusätzliche Überprüfungen und Maßnahmen zu ergreifen, um sicherzustellen, dass alle möglichen Varianten abgedeckt sind.

2.2.1 Normale Enums

Enums funktionieren im einfachsten Fall funktionell und syntaktisch analog zu Rust.

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

2.2.2 Enums mit Werten

Java koppelt zugehörige Werte der Enum-Optionen direkt an die Optionen.

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

    public String label;
    public int weight;

    private Animal(String label, int weight)
        this.label= label;
        this.weight = weight;
    }
}
```

Hier ist zu beachten, dass die Werte der jeweiligen Option Statisch sind. Alle Vögel haben also immer das gleiche Gewicht. Deswegn würde folgender Code auch für die Ausgabe der Werte von e2 die veränderten Werte von e1 ausgeben.

```
public class playground{
        public static void main(String[] args) {
                Animal e1 = Animal.Bird;
                Animal e2 = Animal.Bird;
                e1.weight = 9;
                e1.label = "Newbird";
                System.out.println(e2.label);
                System.out.println(e2.weight);
        }

}
```

Output

```
Newbird
9
```

2.2.3 Enums mit Funktionen

In Java werden Funktionen auf dem Enum im Enum definiert. Abgesehen vom Konstruktor könnte man diese aber auch an einer anderen Stelle definieren.

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

    public boolean isCat(){
        if (this == Animal.Cat){
            return true;
        }else if(this == Animal.Dog){
            return false;
        }else if(this == Anima.Bird){
            return false;
                }else{
            return false;
                }
    }
}
```

2.3 Mächtigkeit von Rust Enums

In diesem Kapitel werfen wir einen genaueren Blick auf die Mächtigkeit von Rust Enums und die Vorzüge, die sie in der Programmierung bieten. Enums sind nicht nur eine einfache Möglichkeit, eine begrenzte Menge von Optionen darzustellen, sondern sie können auch eine Vielzahl von leistungsstarken Konzepten und Techniken bieten. Wir werden die Vorteile von Enums im Detail betrachten und untersuchen, wie sie dazu beitragen können, den Code lesbarer, flexibler und sicherer zu machen. Wir werden die Fülle an Möglichkeiten von Pattern Matching bis hin zur Modellierung komplexer Datenstrukturen erkunden, die Enums in verschiedenen Programmiersprachen bieten.

2.3.1 Das Enum als algebraischer Datentyp

Algebraische Datentypen sind ein Programmierkonzept, das auch in der funktionellen Programmiersprache Haskell verwendet wird und es ermöglicht, komplexe Datenstrukturen auf elegante und präzise Weise zu modellieren. Im gegebenen Beispiel wird die Enum-Struktur Shape als algebraischer Datentyp verwendet. Sie enthält zwei Varianten: Square und Rectangle. Jede dieser Varianten kann verschiedene Informationen oder Daten enthalten, die spezifisch für die jeweilige Form sind. Im Fall des Square wird die Seitenlänge als einziger Parameter übergeben, während im Fall des Rectangle die Länge und Breite als separate Parameter angegeben werden. Die Implementierung der Methode area zeigt, wie algebraische Datentypen in Rust genutzt werden können. Mit dem Pattern Matching-Mechanismus werden die verschiedenen Varianten des Enums überprüft und entsprechende Berechnungen durchgeführt. Wenn das Enum-Objekt self die Square-Variante ist, wird die Fläche des Quadrats berechnet,

indem die Seitenlänge mit sich selbst multipliziert wird. Wenn es sich um die Rectangle-Variante handelt, wird die Fläche des Rechtecks berechnet, indem die Länge mit der Breite multipliziert wird. Das Pattern Matching ermöglicht eine klare und umfassende Abdeckung aller möglichen Varianten, was zu robusterem und sichererem Code führt. In diesem konkreten Beispiel wird die Fläche einer Form berechnet, aber die Anwendungsmöglichkeiten von algebraischen Datentypen in Rust gehen weit über diese einfache Beispiel hinaus. Sie bieten eine flexible und ausdrucksstarke Möglichkeit, Daten zu strukturieren und zu verarbeiten.

```
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,
        }
    }
}
```

2.3.2 Generische Enums

Ein Beispiel für ein generisches Enum in Rust ist das Option-Enum. Das Option-Enum ist äußerst vielseitig und ermöglicht es uns, zwischen zwei Optionen zu wählen: None und Some(T). Die None-Option repräsentiert den Fall, in dem kein Wert vorhanden ist, während Some(T) einen Wert des generischen Typs T enthält. Durch die Verwendung von generischen Typen können wir das Option Enum an verschiedene Datentypen anpassen und eine große Bandbreite von Optionen abbilden.

```
enum Option<T> {
        None,
        Some(T),
}
```

Die interessante Einsatzmöglichkeit des Option Enums ist die Vermeidung von Nullpointer-Exceptionsi, wie im folgendem Beispiel. Falls die Funktion lookUpAnimal etwas findet, wird es mit der some Option zurückgegeben, falls nicht, geben wir None zurück. Mit None kann so der Fall, in dem nichts gefunden wird, explizit behandelt werden.

```rust
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
    }
}
```

2.3.3 Rekursive Enums

In diesem Kapitel geht es um rekursive Enums, die es uns ermöglichen, komplexe Datenstrukturen zu modellieren, die auf sich selbst verweisen. Hier ein einfaches Beispiel mit einem Enum, das mathematische Ausdrücke modelliert. An dieser Stelle ist wichtig, dass die Optionen Plus und Mult das Schlüsselwort Box verwenden. Das ist notwendig, da Enums in Rust eine feste Größe benötigen, um den Speicherbedarf zur Kompilierzeit zu bestimmen. Durch die Verwendung von Box wird ein Zeiger erstellt, der auf die Stelle im Heap verweist, an dem die Option liegt und die tatsächliche Größe des Enum-Objekts auf dem Stack verdeckt. Dies ermöglicht es uns, rekursive Strukturen zu erstellen, ohne dass das Enum eine unendliche Größe haben muss. Indem wir die Variablen left und right als Box<Exp>deklarieren, speichern wir Zeiger auf andere Exp-Objekte im Heap. Dies ermöglicht eine beliebige Verschachtelung von Ausdrücken und die Darstellung von komplexen mathematischen Formeln. Die Verwendung von Box ist daher notwendig, um den rekursiven Bezug in unserem Enum zu realisieren und gleichzeitig eine feste Größe für das Enum-Objekt zur Kompilierzeit zu gewährleisten.

```
pub enum Exp {
    Int {
        val: i32
    },
    Plus {
        left: Box<Exp>,
        right: Box<Exp>
    },
    Mult{
        left: Box<Exp>,
        right: Box<Exp>
    },
}
```

2.3.4 Match Statement

Im folgenden Codebeispiel illustrieren wir die Verwendung des Match-Statements anhand des Exp-Enums in Rust. Das Match-Statement bietet eine effektive Methode, um Enums zu überprüfen und entsprechende Aktionen basierend auf den verschiedenen Optionen auszuführen. Dabei garantiert Rust, dass die Enum-Cases immer abgeschlossen sind, was zu sicherem und zuverlässigem Code führt. Im Beispiel hat jede Option unterschiedliche Felder, die Informationen über den Ausdruck enthalten. In diesem Fall haben wir eine Instanz des Plus-Falls, bei dem die linke und rechte Unterexpression jeweils Int-Expression sind. Die Methode eval wird auf dem Enum-Objekt aufgerufen, um den Ausdruck auszuwerten. Im vorliegenden Fall wird die Plus-Variante erkannt, und das Match-Statement ruft die eval-Methode für die linken und rechten Unterexpressionen auf und addiert die Ergebnisse. Dank der Garantie von Rust, dass die Enum-Cases immer abgeschlossen sind, können wir uns darauf verlassen, dass das Match-Statement alle Fälle abdeckt und somit fehlerfreie Ergebnisse liefert. Dies trägt zur Robustheit und Stabilität unserer Programme bei.

```
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()
       }
   }
}
```

2.3.5 Hinzufügen einer neuen Enum Option

Im gegebenen Codebeispiel haben wir das Exp-Enum aus dem vorherigen Beispiel. Um die Abgeschlossenheit zu illustrieren, wird das Enum nun um die neue Option Div ergänzt. Der Rest bleibt unverändert. Da wir die Div-Variante im Match-Statement nicht abgedeckt haben, wird der Compiler uns einen Fehler melden. Um dieses Problem zu lösen, müssen wir das Match-Statement aktualisieren und die Div-Variante berücksichtigen. Dieses Beispiel zeigt die Stärke und Sicherheit von Rust bei der Arbeit mit Enums. Dadurch erhalten wir robusten und zuverlässigen Code.

```
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

```
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 pattern or an explicit pattern as shown
   |
30 ~            Exp::Mult{left, right} => left.eval() * right.eval(),
31 +            &Exp::Div { .. } => todo!()
   |

For more information about this error, try `rustc --explain E0004`.
```

2.3.6 Nested Pattern Matching

In diesem Codebeispiel wird das Konzept von Nested Pattern Matching illustriert. Innerhalb des Match-Statements für die Mult-Variante führen wir ein weiteres Match-Statement aus, um die linke Unterexpression zu überprüfen. Wir verwenden das Doppelsternchen-Operator (**left), um auf den Inhalt der Box zuzugreifen. Innerhalb des inneren Match-Statements überprüfen wir, ob die linke Unterexpression ein Int-Objekt den Wert 0 hat. Wenn dies der Fall ist, wird sofort der Wert 0 zurückgegeben, da das Ergebnis der Multiplikation mit 0 immer 0 ist und wir uns so das Evaluieren des rechten Ausdrucks sparen. Wenn die linke Unterexpression kein Int-Objekt mit dem Wert 0 ist, wird die Multiplikation der linken und rechten Unterexpressionen durchgeführt und das Ergebnis zurückgegeben. Die Verwendung von Nested Pattern Matching ermöglicht es uns, präzise Logik zu implementieren und spezifische Aktionen basierend auf verschiedenen Kombinationen von Varianten auszuführen. Es hilft auch, den Code lesbar und verständlich zu halten. Die Alternative zu Nested Pattern Matching wäre eine If/Else-Abfrage. Diese werden aber mit wachsender Komplexität der Abfrage sehr verbos und fehleranfällig.

```rust
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()
```

```
            }
        }
    }
}
```

2.3.7 Erweiterbare Funktionen für Enums

Mittels des impl Blocks ist es sehr einfach, neue Funktionalitäten für Enums (oder auch Structs) bereitzustellen, ohne diese selbst zu ändern. Im Beispiel fügen wir die neue Funktion treeHeight zum Enum hinzu, die die Tiefe unseres Expression-Trees errechnet.

```
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 Funktionalität von Rust Enums in Java

In diesem Kapitel werden wir uns mit der Herausforderung befassen, wie man die Funktionalität von Rust Enums in der Programmiersprache Java nachbilden kann. Eine Möglichkeit, Rust Enums in Java nachzubauen, besteht darin, Klassen zu erstellen, die verschiedene Optionen repräsentieren. Diese Klassen können spezifische Eigenschaften und Methoden haben, um das Verhalten der einzelnen Optionen zu definieren. Durch den Einsatz von Vererbung und Polymorphie können wir eine ähnliche Struktur und Flexibilität wie in Rust Enums erreichen. Ein weiterer wichtiger Aspekt von Rust Enums ist das abgeschlossene Pattern Matching. Dies bedeutet, dass jede mögliche Option des Enums in den entsprechenden Match-Blöcken abgedeckt werden muss. In Java können wir versuchen, eine ähnliche Funktionalität durch die Verwendung von Switch-Anweisungen und Fall-Through-Fällen zu erreichen. Es ist zu beachten, dass Fall-Through-Fälle eventuell trotzdem zu ungeplanten Ergebnissen führen können. Es ist wichtig anzumerken, dass Java und Rust unterschiedliche Programmiersprachen sind und jeweils ihre eigenen Stärken und Schwerpunkte haben. Daher kann es Herausforderungen geben, wenn man versucht, die volle Funktionalität von Rust Enums in Java nachzubilden. Dennoch bietet dieses Kapitel die Möglichkeit, kreative Lösungen zu finden und die Vorteile von Rust Enums auf Java-Anwendungen zu übertragen.

2.4.1 Expression-Logik in Java

Der Beispielcode zeigt das vorherige Beispiel mit den mathematischen Ausdrücken, diesmal aber in Java mit Klassen realisiert. Da jeder Ausdruck eine andere Form hat, ist dies notwendig, um die Struktur zu beschreiben. Realisiert wurde es konkret durch das Benutzen einer abstrakten Klasse, die eine Implementation der eval-Funktion fordert. An dieser Stelle ist interessant, dass kein Switch-Case benutzt wird, da die eval-Funktionen mit ihrem entsprechenden Typ fest gekoppelt sind. Switch-Case wäre an dieser Stelle, aus den oben genannten Gründen, für eine Typunterscheidung gar nicht in der Lage, die unterschiedlichen Klassen zu unterscheiden. Falls man die eval-Funktion aus den Klassen bspw. auslagern möchte, müsste man für die Unterscheidung mit dem instanceof Operator alle möglichen Fälle abdecken.

```
public class Expression {
    public static void main(String[] args) {
        PlusExp e = new PlusExp(new IntExp(5), new IntExp(10));
        System.out.println("Der Ausdruck evaluiert zu: "+e.eval());
    }
}

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 Match Statement in Java

In Java ist das Switch Case-Konstrukt eine Möglichkeit, verschiedene Fälle basierend auf einem Wert auszuwerten und entsprechenden Code auszuführen. Es ermöglicht das Vergleichen eines Werts mit verschiedenen Konstanten. Allerdings hat der Switch Case in Java einige Einschränkungen verglichen mit Match:

- Die Verwendung von Switch Case ist auf primitive Datentypen oder Enums beschränkt. Es ist nicht möglich, komplexe Muster oder Strukturen effizient abzubilden.
- Die Veränderung des Enums, das für den Switch Case verwendet wird, kann zu unerwartetem Verhalten führen. Der Compiler warnt nicht vor vergessenen Fällen oder nicht abgedeckten Werten.

Des Weiteren können in Java komplexe Muster und verschachteltes Pattern Matching nur durch weitere If/Else-Abfragen erreicht werden. Dies führt oft zu umständlichem und schwer lesbarerem Code. All das sind Eigenschaften, die sich auch nicht über cleveres Ausnutzen von Datenstrukturen oder Designpatterns

lösen lassen. Im Beispielcode findet man eine Abfragelogik, die via Switch-Case implementiert ist.

```
public class AnimalSwitchCase {
    public static void main(String[] args) {
        Animal a = Animal.Bird;
        switch (a){
            case Dog:
                System.out.println("It's a Dog");
                break;
            case Bird:
                System.out.println("It's a Bird");
                break;
            case Cat:
                System.out.println("It's a Dog");
                break;
            default:
                System.out.println("This Animal wasn't expected");
        }
    }
}
enum Animal{
    Dog,
    Cat,
    Bird
}
```

Pattern Matching und Nested Pattern Matching sind an sich nur durch explizite If/Else Abfragen möglich. Da Enums sich aber nicht verändern und die Werte bei jeder Option gleich sind, ist das einzige Pattern, das man unterscheiden könnte, die Option des Enums selbst. Eine genauere Unterscheidung tritt bei Klassen auf.

3 Traits

Traits behandeln eine zentrale Funktion in Rust, mit der Entwickler Funktionalität zwischen verschiedenen Typen teilen können. Traits werden oft mit Interfaces in anderen Programmiersprachen verglichen, sind jedoch keine direkten Äquivalente. Ein Trait definiert eine Menge von Funktionen, die auf einen bestimmten Typ angewendet werden können. Es ermöglicht die gemeinsame Nutzung von Verhaltensweisen über verschiedene Typen hinweg. Im Wesentlichen legt ein Trait fest, welche Funktionen ein Typ bereitstellen muss, um als implementierendes Objekt des Traits zu gelten. Im Vergleich zu Interfaces sind Traits flexibler und leistungsfähiger. Während Interfaces in anderen Sprachen hauptsächlich die Schnittstelle eines Typs beschreiben, gehen Traits weiter und beschreiben eine Menge von Funktionen, die auf den Typ angewendet werden können. Traits können daher als Prädikat für einen Typ betrachtet werden, bei dem ein Typ nur akzeptiert wird, wenn er die Funktionen des Traits implementiert, wohingegen Interfaces ein eigener Typ sind. Traits ermöglichen es Entwicklern, gemeinsame Funktionalitäten zu abstrahieren und wiederzuverwenden, indem sie definiert und auf verschiedene Typen angewendet werden. Dadurch wird der Code modularer und flexibler.

3.1 Traits in Rust

3.1.1 Einfacher Trait

Der Trait Shape fungiert hier als Prädikat für die implementierenden Strukturen. Das bedeutet, dass eine Struktur nur dann als implementierendes Objekt des Traits gilt, wenn sie die Methode area implementiert. Die Methode area wird jeweils für die Berechnung der Fläche des Quadrats und des Rechtecks implementiert. Durch die Implementierung des Traits Shape können wir nun auf die Methode area für Instanzen von Square und Rectangle zugreifen, unabhängig von der spezifischen Struktur. Dies ermöglicht es uns, die gleiche Funktionalität für verschiedene Formen zu verwenden und den Code zu vereinfachen. Wenn eine Struktur den Trait Shape implementiert, garantieren wir, dass sie über die Methode area verfügt und somit als gültige geometrische Form betrachtet werden kann. In diesem Beispiel ermöglicht der Trait Shape die Bereitstellung einer allgemeinen Funktionalität zur Berechnung der Fläche, die von verschiedenen geometrischen Formen verwendet werden kann.

```
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
    }
}
```

3.1.2 Shorthand Schreibweise

Durch diese Schreibweise ist es möglich, die Funktionen wie auf einem Objekt aufzurufen. In beiden Fällen ist der erste Parameter der Funktion einfach die Instanz des Typen selbst.

```
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());
}
```

3.1.3 Default-Implementationen

Traits erlauben es, wie Interfaces und abstrakte Klassen, ein Standartverhalten bereitzustellen, das nicht expliziert implementiert werden muss. Das heißt mit der Zeile 'impl Animal for Cat' gilt das Prädikat schon als erfüllt.

```
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{}
```

Führt man das Programm aus, so erhält man Folgendes.

Output:

```
The Animal made a noise
```

3.1.4 Trait Bounds

Beim Definieren von Funktionen können wir die Traits benutzen, um Typen einzuschränken. Diese Einschränkung nennt man auch Trait Bound. Im Beispiel wird vom Typen A und B verlangt, dass sie das Prädikat Shape erfüllen, um sicherzugehen, dass sie Zugriff auf die notwendige Area-Methode haben.

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

3.1.5 Multiples Binding

Es ist auch möglich, mehrere Anforderungen an einen Typ zu stellen. Dies kommt der logischen Verundung von Prädikaten gleich. Sprich ein Typ A wird nur aktzeptiert, wenn es eine Implementation von Shape und OtherTrait für diesen Typen gibt.

```
fn someFunc<A:Shape+OtherTrait>(x : &A){
        ...
 }
```

3.1.6 Dynamische Traits

Dynamische Traits sind äquivalent zu Interfaces in Java. Box ist notwendig, da unsere Typen, die den Trait implementieren, theoretisch beliebig groß sein können. Dies erlaubt es uns, konkrete Typen als Parameter und Rückgabewerte anzugeben, die eine Implementation für den Trait haben müssen.

```
fn sum_area(x : Box<dyn Shape>, y: Box<dyn Shape>) -> i32 {
    return area(x) + area(y)
 }
```

3.1.7 Kurzschreibweise für dynamische Traits

Oft sieht man auch diese etwas kürzere Schreibweise, die die Verwendung von Box für uns versteckt.

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

3.1.8 Platzhaltertypen

Platzhaltertypen ermöglichen es uns, für jede Implementierung eines Traits eine Reihe von Typen für die Implementation festzulegen. Im Beispiel von TransformAB gibt man den Typen A an, der über eine Funktion in Typ B umgewandelt wird. Dies ist auch über generische Traits möglich, mit dem Unterschied, dass man generische Traits mehrmals implementieren kann.

```
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)
    }
}
```

3.1.9 Assoziierte Konstanten

Ähnliche wie bei Platzhaltertypen kann man für jede Implementierung des Traits gewisse Konstanten festlegen. In diesem erweiterten Beispiel des TransformAB Traits findet die Umformung in einen Vektor statt, der das umgewandelte Struct so oft enthält, wie man es in der Konstante angegeben hat. Es ist wichtig anzumerken, dass es keine assoziierten Variablen gibt. Nur Konstanten sind erlaubt.

```
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
    }
}
```

3.1.10 Supertraits

Supertraits ermöglichen es uns, Traits zu definieren, die von anderen Traits erben. Dadurch können wir eine Hierarchie von Traits erstellen und sicherstellen, dass eine Struktur alle Anforderungen der vererbten Traits erfüllt. In dem gegebenen Beispiel werden mehrere Traits definiert, darunter Person, Student, Programmer und CompSciStudent. Der Trait CompSciStudent erbt von den Traits Programmer und Student, was bedeutet, dass eine Struktur, die den Trait CompSciStudent implementiert, auch eine Implementation der anderen Traits erfüllen muss. Die Implementierungen der Methoden für die Traits verwenden die Felder der Struktur, um die entsprechenden Werte zurückzugeben. So ist es möglich, mit einem einzelnen Trait Bound eine personalisierte Ausgabe zu machen. Supertraits ermöglichen es uns, klassenhirarchisch ähnliche Strukturen aufzubauen.

```
fn main() {
    let s = HskaStudent{name:"Mario", university:"hska", fav_language:"rust", git_username
    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
}
struct HskaStudent{
    name: &'static str,
    university: &'static str,
    fav_language: &'static str,
```

```
    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)
    }
}
```

3.2 Mächtigkeit von Traits

Wie bereits erwähnt besitzen Traits gewisse Vorteile gegenüber Interfaces, die ich nun anhand einiger einfacher Beispiele erläutern werde. Man sollte anmerken, dass noch mehr Vorteile gibt als die, die hier aufgeführt sind.

3.2.1 Gleiche Methodensignatur

Mit Traits ist es möglich, mehrere geteilte Funktionalitäten über einem Typ zu bilden, selbst wenn diese die gleiche Signatur haben. Das Beispiel unten würde sich über Interfaces nicht einfach so implementieren lassen, da ein Signaturkonflikt entstehen würde. Dazu aber später mehr. Beim Methodenaufruf wird man jedoch dazu gezwungen, auf die Kurzschreibeweise zu verzichten, da der Aufruf sonst ambig wäre.

```
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);
}

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");
  }
}
```

3.2.2 Generische Mehrfachimplementierung

Traits erlauben es uns, den selben Trait mehrfach mit unterschiedlichen generischen Parametern zu implementieren. In Java ist dies mit Interfaces nicht möglich. Später werden wir noch sehen, wie wir auch dieses Problem in Java lösen können. Da Rust nur Funktionsmengen definiert, gibt es keinen Konflikt, solange eindeutig ist, welche Funktion beim Aufruf benutzt wird.

```
fn main() {
    let s : some_struct = some_struct{};
    let someInteger: i32 = s.mygenval();
    let someString: String = s.mygenval();
}
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()
```

```
    }
}
```

3.2.3 Referenzierung des eigenen Typen

Traits sind in der Lage, mittels Self den Typen zu referenzieren, für den man sie implementiert. Dies ist wie im Beispiel Clone wichtig, da wir den eigenen Typen für beliebe Structs zurückgeben möchten.

```
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};
    }
}
```

3.2.4 Funktionalität für Third-Party-Datentypen

Ein Problem bei Java ist auch, dass man nicht ohne weiteres Funktionalität externer Klassen erweitern kann, ohne diese zu verändern. Würden wir eine Dependency importieren, bei der wird möchten, dass sie eine sleep Methode unterstützt, ist dies nicht möglich, ohne die Dependency selbst zu verändern. Bei Traits implementiert man einfach für den fremden Datentyp mittels des impl Blocks den gewünschten Trait. Hier ein Beispiel:

```
use std::thread;
use std::time::Duration;

fn main() {
    thiryparty_struct{}.sleep();
}
struct thiryparty_struct{}

trait Sleep{
```

```
    fn sleep(&self);
}

impl Sleep for thiryparty_struct {
    fn sleep(&self){
        thread::sleep(Duration::from_millis(1000));
    }
}
```

3.2.5 Konditionelle Implementierung

Via Trait Bounds ist Rust in der Lage, Implementationen nur für Teilmengen von Structs zu definieren, falls diese von einem generischen Parameter abhängig sind. Im Beispielcode wird dies mittels des Pair Structs illustriert. Dieses stellt die Methode cmp_display zur Verfügung, wenn T eine Implementation für den Display und PartialOrd Trait besitzt. Dies ist in Java unmöglich. Wir könnten nur T komplett beschränken und erlauben, dass es nur Paare von Typen geben kann, die ausgebbar und partiell geordnet sind.

```
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);
        }
    }
}
```

3.3 Traitfunktionalität in Java

Es ist möglich, manche der Funktionalitäten von Rust Traits in Java zu haben. Designpatterns[4] helfen uns an dieser Stelle. Gewisse Funktionalitäten wie konditionelle Implementierung sind aber nicht möglich.

3.3.1 Gleiche Methodensignatur

In Java ist es nicht möglich, mehrere Interfaces zu implementieren, die eine gleiche Signatur haben. Bei Vererbung kann dies nicht passieren, da man immer nur eine super-klasse hat. Falls man die gleiche methode noch einmal in der Unterklasse implementiert, wird damit die Methode aus der Superklasse überschrieben und ist damit nicht ambig. Hier noch einmal das Beispiel von oben in java. Dies führt zu einem Fehler bei der Kompilierzeit.

```
class someclass implements musicplayer, boardgame{
   public void play(){
       system.out.println("you are playing");
   }
}
interface musicplayer{
   public void play();
}
interface boardgame{
   public void play();
}
```

Lösung: Falls wir aber trotzdem beide Schnittstellen anbieten müssen, hilft uns das sogenannte Adapter-Pattern[4]. Es stellt die gewünschte Schnittstelle anstelle der eigentlichen Klasse zur Verfügung und managt den Methodenaufruf.

```
public class adaptercompatible {
    public static void main(string[] args) {
        someclass sc = new someclass();
        musicplayeradapter ma = new musicplayeradapter(sc);
        boardgameadapter ba = new boardgameadapter(sc);
        ma.play();
        ba.play();
    }
}
class someclass{
    public void playmusic(){
        system.out.println("playing music");
    }
    public void playBoardGame(){
        System.out.println("Playing boardgame");
    }
}
interface MusicPlayer{
    public void play();
}
interface BoardGame{
    public void play();
}

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();
    }
}
```

Der Text wird durch das Benötigen von mehreren Klassen viel verboser und unübersichtlicher. Des weiteren muss man sich mit Design Pattern auseinandersetzen, was Insgesamt für sehr viel Aufwand für die gleiche Funktionalität sorgt.

3.3.2 Generische Mehrfachimplementierung

Ähnlich wie Oben gibt es einen Konflikt zwischen den Interfaces, die man implementieren möchte. Das konkrete Problem hier ist, dass man das gleiche Interface nicht mehrmals implementieren darf, selbst wenn der generische Parameter unterschiedlich ist und es keinen Signaturkonflikt gibt.

```
public class SomeClass implements Generic<Integer>, Generic<String> {
    public static void main(String[] args) {
        SomeClass sc = new SomeClass();
    }
}

interface Generic<T> {
    public T mygenvalue();
}
```

Output:

```
SomeClass.java:1: error: Generic cannot be inherited with
different arguments: <java.lang.Integer> and <java.lang.String>
```

Lösung: Verwenden wir aber jeweils einen Adapter, der das jeweilige generische Interface implementiert, tritt das Problem nicht auf. An dieser Stelle ist zu erwähnen, dass generell alle Probleme, die durch einen Konflikt zweier Interfaces auftreten, sich über das Adapter-Pattern lösen lassen.

```
public class SomeClass {
    public static void main(String[] args) {
        SomeClass sc = new SomeClass();
        Integer someInt = new GenericIntAdapter(sc).mygenvalue();
        String someString = new GenericStringAdapter(sc).mygenvalue();
    }
}

interface Generic<T> {
    public T mygenvalue();
}

class GenericIntAdapter implements Generic<Integer> {
    private SomeClass someClass;

    public GenericIntAdapter(SomeClass someClass) {
        this.someClass = someClass;
    }

    @Override
    public Integer mygenvalue() {
        return 5;
    }
}
class GenericStringAdapter implements Generic<String> {
    private SomeClass someClass;

    public GenericStringAdapter(SomeClass someClass){
        this.someClass = someClass;
    }
    @Override
    public String mygenvalue() {
        return "abc";
    }
}
```

3.3.3 Funktionalität für Third-Party-Datentypen

Da bei Java die Klasse und ihre Funktionen sehr stark gekoppelt sind, ist es nicht ohne weiteres möglich, eine Funktionalität bereitzustellen, ohne die Klasse selbst zu verändern. Das Wrapper-Pattern[4] erlaubt es uns, eine Hülle um die eigentliche Klasse zu legen, die dann die eigentliche Funktion zur Verfügung stellt. Falls man in diesem Kontext mit dem Objekt der umhüllten Klasse interagieren möchte, geschieht der Zugriff nur indirekt über die Wrapper-Klasse. Im Beispiel stellen wir der Klasse ThirdParty mittels Wrapper-Klasse indirekt die zusätzliche Funktionalit von Sleep zur Verfügung.

```java
public class ThirdParty {
    public static void main(String[] args) {
        ThirdParty original = new ThirdParty();
        WrapperClass wrapper = new WrapperClass(original);

        wrapper.doSomething();
        wrapper.sleep();
        wrapper.doSomething();
    }
    public void doSomething() {
        System.out.println("Doing something...");
    }
}

class WrapperClass {
    private ThirdParty original;

    public WrapperClass(ThirdParty original) {
        this.original = original;
    }

    public void doSomething() {
        original.doSomething();
    }

    public void sleep() {
        try {
            Thread.sleep(1000); // Sleep for 1000 milliseconds
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

3.3.4 Referenzierung des eigenen Typen

Dieses Problem wird häufig mittels generischer Interfaces oder Klassen gelöst. Als generischen Typen kann man einfach die Klasse selbst übergeben. Der einzige Nachteil ist, dass die Information redundant erscheint.

```java
public class SameInputOutput{
    public static void main(String[] args) {

    }
}
interface sameObject<T>{
    public T returnSameObject(T input);
}

class Dog implements sameObject<Dog>{
```

```
    public Dog returnSameObject(Dog input){
        return input;
    }
}
```

3.3.5 Konditionelle Implementierung

Wie bereits im Kapitel über Rusts konditionelle Implementierung erwähnt, ist diese in Java nicht möglich. Es ist lediglich möglich, den generischen Parameter selbst zu beschränken. Wenn wir wie im Beispiel das Hunde-Paar instanziieren möchten, bekommen wir einen Fehler zur Kompilierzeit.

```
public class Conditional {
    public static void main(String[] args) {
        Pair<Integer> intpair = new Pair(1,2);
        Pair<Dog> dogpair = new Pair(new Dog("Ben"), new Dog("Albert"));
    }
}

class Pair<T extends Comparable<T>>{
    private T x;
    private T y;

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

    public void cmpDisplay() {
        if (x.compareTo(y) >= 0) {
            System.out.println("The largest member is x = " + x);
        } else {
            System.out.println("The largest member is y = " + y);
        }
    }
}

class Dog{
    private String name;
    public Dog(String name){
        this.name = name;
    }
}
```

Output:

```
Conditional.java:4: error: type argument Dog is not within bounds of type-variable T
Pair<Dog> dogpair = new Pair(new Dog("Ben"), new Dog("Albert"));
          ^
where T is a type-variable:
T extends Comparable<T> declared in class Pair
```

```
Conditional.java:4: error: incompatible types: Dog cannot be converted to Comparable
Pair<Dog> dogpair = new Pair(new Dog("Ben"), new Dog("Albert"));
```

Es ist jedoch möglich, über Vererbung unterschiedliche Typen von Pair darzustellen, bei denen die Unterklassen einen eingeschränkten generischen Parameter T hat. Im Beispielcode unten aktzeptiert Pair alle Parameter T. Die Klasse SpecifiedPair jedoch aktzeptiert nur Klassen, die das Interface 'IGiveHelloString' implementieren.

```
public class PairExample2
    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(){
    }
}

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

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

```
}

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

4 Vergleich

4.1 Enums

4.1.1 Rust Enums Vorteile

- Algebraische Datentypen: Da Rust Enums algebraische Datentypen sind, erlauben sie es uns unterschiedliche Strukturen für die Varianten anzugeben. Obwohl die Datentypen so unterschiedlich sind, macht das Verwenden des Enums, die Beziehung, in der sie zueinander stehen deutlich.
- Starke Typsicherheit: Der Compiler stellt sicher, dass alle Varianten eines Enum im Pattern Matching abgedeckt sind. Da jeder Fall genau abgedeckt ist können hiermit Laufzeitfehler vermieden werden.
- Nested Parrern Matching: Mit Nested Pattern Matching lassen sich komplexere Typen, die Referenzen auf andere Datentypen oder gar sich selber verwalten, sehr präzise zerlegen und dem entsprechenden Fall zuordnen.
- Erweiterbarkeit der Funktionalität eines Enums: Es ist möglich Funktionen die auf einem Enum arbeiten, mittels impl Blöcken zu erweitern. Die Funktionalität ist somit zwar unabhängig vom Enum aber doch direkt zugeordnet. Dies hilft den Code zu organisieren und flexibel zu halten.
- Generics: Enums in Rust können generisch sein, was bedeutet, dass sie Typ-Parameter haben können. Dies ermöglicht wiederverwendbaren und flexiblen Code.

4.1.2 Rust Enums Nachteile

- Hinzufügen und Entfernen von Varianten: Das Hinzufügen einer neuen Variante erfordert oft eine Überprüfung und Anpassung des gesamten Codes, der das Pattern Matching verwendet.
- Komplexität des Pattern Matchings: Das Pattern Matching mag für einfache Fälle sehr übersichtlich sein, jedoch kann die Verwendung von Pattern Matching für sehr komplexe Hierarchien zu einer erhöhten Komplexität und Unübersichtlichkeit führen. Wenn sich die Varianten eines Enums verändern, ist es oft kompliziert das Pattern Matching entsprechend anzupassen.

4.1.3 Java Klassen und virtuelle Methoden Vorteile

- Erweiterbarkeit: Es ist einfacher, neue Klassen hinzuzufügen, ohne den bestehenden Code zu ändern.
- Funktionalität teilbar: Durch Klassen ist es möglich implementierte Methode Methoden an andere Klassen zu vererben. Dadurch lässt sich Redundanz des Codes vermeiden.
- Funktionalität an Klassen gekoppelt: Da die Funktionalität an die Klasse direkt gekoppelt ist, teilt sich die Komplexität der Logik auf die einzelnen Klassen auf.

4.1.4 Java Klassen und virtuelle Methoden Nachteile

- Mehr Boilerplate: Java Klassen haben viel Boiletplate Code. Dies macht den Code an sich oft unnötig verbos, für die Logik die er abbildet.
- Typfehler zur Laufzeit: Das Typsystem von Java ist weniger streng als das von Rust, was zu Fehlern führen kann.

4.2 Traits

4.2.1 Traits Vorteile

- Beliebige funktionelle Erweiterung von Structs: Mit Traits in Rust kann man Methoden zu Structs hinzufügen, ohne den ursprünglichen Code zu verändern.
- Erweiterung von 3rd Party Structs: Da wir nur Funktionsmengen für Typen definieren ist es egal ob wir direkten ZUgriff auf den Typen haben oder nicht. Deswegen können wir auch für Typen, die wir nicht selbst geschrieben habe Funktionalität und Verhalten definieren.
- Konditionelle Implementierung mit Trait Bounds möglich: Es ist möglich Funktionalität nur dann zu implementieren, wenn bestimmte Trait-Bounds erfüllt werden. Das ist spezielle Funktionalität für bestimmte Teilmengen des Typens bereitstellen möchte, den Typ selber aber allgemein halten will.

4.2.2 Traits Nachteile

- Implementation von Traits zwischen Typen nicht teilbar: Es ist nicht möglich die Implementation eines Traits mit einem Trait zu teilen der von ihm erbt. Der Untertrait muss trotzdem seine eigene Implementation schreiben um das Prädikat des Traits zu erfüllen. Dies führt ggf. zu redundantem Code.
- Keine statischen Konstanten zwischen Implementationen: Es ist nicht möglich dass es Konstanten gibt, auf die alle Implementation des Traits zugreifen können

4.2.3 Interfaces Vorteile

- Vererbung von Datenfeldern und Funktionen möglich.
- Überschreiben von Funktionen in Unterklasse möglich.

4.2.4 Interfaces Nachteile

- Konflikte zwischen Interfaces: Oft kommt es vor, dass sich zwei Interfaces nicht gleichzeitig implementieren lassen. Um die Funktionalität beider zu gewährleisten muss Java auf Designpatterns zurückgreifen, die oft verbos sind.

- Funktionelle Erweiterung nur durch Veränderung der Klasse selbst: Da Funktionen und Klassen nah gekoppelt sind ist es nur möglich weitere Funktionalität für eine Klasse zu implementieren, wenn man diese Verändert. Um Funktionalität für 3rd-Party-Datentypen zu gewährleisten muss Java an dieser Stelle wieder auf Designpatterns zurückgreifen, die sehr verbos sind.
- Viel Boilerplate: Java ist auch ohne Designpatterns durch sehr viel Boilerplate-Code sehr verbos.

Literatur

[1] Slides about haskell by prof. dr. sulzmann. `https://sulzmann.github.io/ProgrammingParadigms/lec-rust-vs-haskell.html`.

[2] The rust programming language. `https://doc.rust-lang.org/stable/book/`.

[3] Slides about rust by prof. dr. sulzmann. `https://sulzmann.github.io/ProgrammingParadigms/lec-rust.html`.

[4] Design patterns. `https://refactoring.guru/design-patterns`.