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. Die Konzepte einige Vorteile, gegenüber normalen Enums in Java sowie Interfaces und Vererbung, die versuchen ähnliche Probleme zu lösen. 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 Sprache wie Java aber auch gar nicht bereitstellen. Bezogen auf diese Konzepte ist Rust mit Java verglichen also syntaktisch prägnanter, lesbarer und stellenweise funktinell 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 robuster wird. 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 über einem Typen und erlauben es, bestimmte Verhaltensweisen zu 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 in der Programmierung eine Möglichkeit, eine begrenzte Anzahl von Varianten oder Optionen zu definieren. Sie dienen dazu, eine feste Menge von Werten darzustellen, die ein bestimmtes Konzept oder eine bestimmte Entität repräsentieren können. Enums ermöglichen es, den Code lesbarer zu gestalten, indem sie eine klar definierte Liste von möglichen Zuständen oder Varianten zur Verfügung stellen. Auf den ersten Blick mögen Enums in Rust und Java ähnlich aussehen. Beide Sprachen bieten Mechanismen, um eine begrenzte Anzahl von Optionen zu modellieren. 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 numerische Werte hinausgehen. Sie können Varianten enthalten, die selbst wiederum Daten oder weitere Optionen beinhalten können. Diese Flexibilität eröffnet ganz neue Möglichkeiten bei der Modellierung von Daten. Ein weiteres starkes 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 sicherzustellen, dass alle Fälle behandelt werden. Diese Funktionalitäten machen Rust Enums zu einer mächtigen und robusten Technik, die über die Möglichkeiten von Java Enums hinausgeht.

Anhand einfacher Beispiele gehen zeigen ich zunächst, wie simple Enums, Enums mit zugehörigen Werten und Enums mit Funktionen in Rust und Java aussehen. Danach gehe ich auf den konkreten Unterschied 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

2.1.1 Normale Enums

Einfache Auflistungen sind in Rust und Java komplett identisch.

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

2.1.2 Enum 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, dies 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 Enum mit Funktionen

Das definieren von Funktionen über einem Enumtypen funktioniert analog wie mit den 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

- Enums sind spezielle Klasse
- Enumtypen sind Instanzen
- Instanz statisch und final (per default)

In Java werden Enums als spezielle Klassen behandelt. Sie dienen dazu, eine fest definierte Menge von Optionen oder Zuständen zu repräsentieren. Im Gegensatz zu herkömmlichen Klassen erlaubt die Verwendung von Enums eine klarere und lesbarere Darstellung von möglichen Werten. 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 in der Regel als statisch und final deklariert werden. Dies bedeutet, dass die Werte der Instanzvariablen für jede einzelne Instanz des Enums gleich sind und nicht verändert werden können. Dadurch wird sichergestellt, dass Enums unveränderliche Optionen repräsentieren und konsistent bleiben. Durch die Verwendung von speziellen Klassen und Instanzen ermöglichen Java Enums eine effiziente und sichere Handhabung einer begrenzten Menge von Optionen. Die statischen und finalen Instanzvariablen gewährleisten dabei eine konsistente und unveränderliche Repräsentation der 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. Trotzdem bieten Java Enums eine solide Grundlage für die Modellierung und Verwendung einer begrenzten Anzahl von Optionen in Java-Programmen. Sie ermöglichen eine lesbarere und intuitivere Darstellung von Zuständen und Optionen und tragen so zur Verbesserung der Codequalität und -wartbarkeit bei.

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 für die Enum-Optionen direkt an die Option selber.

```
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 Enum 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. Interessant ist auch, dass wir bei der Abfrage anders als beim Pattern-Matching immer einen default Case der Abfrage brauchen, da Java zur Compile-Zeit schon nicht weiß, dass die Funktion schon wohl definiert ist. Hierzu aber später mehr.

```
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 nächsten Kapitel werfen wir einen genaueren Blick auf die Mächtigkeit von 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. Von den vielfältigen Anwendungsmöglichkeiten von Pattern Matching bis hin zur Modellierung komplexer Datenstrukturen - wir werden die Fülle an Möglichkeiten erkunden, die Enums in verschiedenen Programmiersprachen bieten. Seien Sie gespannt auf die spannenden Aspekte, die wir in Bezug auf die Mächtigkeit von Enums beleuchten werden und wie sie Ihr Verständnis und Ihre Fähigkeiten in der Programmierung erweitern können.

2.3.1 Der Enum als algebraischer Datentyp

Algebraische Datentypen sind ein leistungsstarkes Konzept in der Programmierung, das es ermöglicht, komplexe Datenstrukturen auf elegante und präzise Weise zu modellieren. In Rust werden Enums als algebraische Datentypen betrachtet, da sie die Möglichkeit bieten, eine begrenzte Anzahl von Varianten oder Optionen zu definieren. Im gegebenen Beispiel wird die Enum-Struktur SShapeäls algebraischer Datentyp verwendet. Sie enthält zwei Varianten: SSquareünd

Rectangle". Jede dieser Varianten kann verschiedene Informationen oder Daten enthalten, die spezifisch für die jeweilige Form sind. Im Fall des SSquare"wird die Seitenlänge als einziger Parameter übergeben, während im Fall des RRectangle"die Länge und Breite als separate Parameter angegeben werden. Die Implementierung der Methode äreaßeigt, wie algebraische Datentypen in Rust genutzt werden können. Mit dem Pattern Matching-Mechanismus werden die verschiedenen Varianten der Enum überprüft und entsprechende Berechnungen durchgeführt. Wenn das Enum-Objekt ßelf"die SSquareVariante ist, wird die Fläche des Quadrats berechnet, indem die Seitenlänge mit sich selbst multipliziert wird. Wenn es sich um die RectangleVariante handelt, wird die Fläche des Rechtecks berechnet, indem die Länge mit der Breite multipliziert wird. Durch die Verwendung von algebraischen Datentypen in Rust können komplexe Datenstrukturen einfach und übersichtlich modelliert werden. 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 Illustration 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

In diesem aufregenden Kapitel werden wir uns mit generischen Enums in Rust beschäftigen, die eine leistungsstarke Möglichkeit bieten, flexible und parametrisierte Datenstrukturen zu modellieren. Ein herausragendes Beispiel für ein generisches Enum in Rust ist das Öption¡T¿Enum. Das Öption¡T¿Enum ist äußerst vielseitig und ermöglicht es uns, zwischen zwei Optionen zu wählen: Noneünd SSome(T)". Die NoneOption repräsentiert den Fall, in dem kein Wert vorhanden ist, während SSome(T)ëinen Wert des generischen Typs Tënthält. Durch die Verwendung von generischen Typen können wir das Öption¡T¿Enum an verschiedene Datentypen anpassen und eine große Bandbreite von Optionen

abbilden. Das Beispiel verdeutlicht die Anwendung der generischen Enums anhand des Öption¡T¿Enums. Es bietet eine elegante Lösung für Situationen, in denen ein Wert optional sein kann. Durch die Verwendung des Öption¡T¿Enums können wir klar ausdrücken, dass ein Wert entweder vorhanden ist (durch SSome(T)") oder nicht vorhanden ist (durch None"). Die Verwendung von generischen Enums wie dem Öption¡T¿Enum bietet mehrere Vorteile. Sie ermöglichen eine typsichere Behandlung von Optionen, da der zugrunde liegende Typ T"bei der Kompilierung überprüft wird. Dadurch wird vermieden, dass unbehandelte Fälle auftreten und Fehler durch den Zugriff auf nicht vorhandene Werte entstehen. Das Pattern Matching ist ein mächtiges Werkzeug, um generische Enums zu verarbeiten. Es erlaubt uns, die verschiedenen Optionen des Enums zu überprüfen und entsprechende Aktionen durchzuführen. Wir können leicht feststellen, ob eine Option Noneïst und dementsprechend reagieren, oder den Wert einer SSome(T)Option extrahieren und damit weiterarbeiten. Durch die Verwendung von generischen Enums eröffnen sich in Rust vielfältige Möglichkeiten zur Modellierung und Verarbeitung von Daten. Sie bieten Flexibilität und Parametrisierung, um verschiedene Datentypen zu unterstützen und maßgeschneiderte Lösungen zu ermöglichen. Im nächsten Kapitel werden wir noch tiefer in die Vorzüge und Anwendungsfälle generischer Enums eintauchen und ihre Stärken voll ausschöpfen.

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

2.3.3 Rekursive Enums

In diesem spannenden Kapitel werden wir uns mit rekursiven Enums in Rust beschäftigen, die es uns ermöglichen, komplexe Datenstrukturen zu modellieren, die auf sich selbst verweisen. Ein Beispiel für ein rekursives Enum ist das ËxpEnum. Das ËxpEnum repräsentiert mathematische Ausdrücke und kann verschiedene Varianten enthalten. Es gibt eine Variante Ïnt", die einen einzelnen Ganzzahlwert enthält. Darüber hinaus gibt es die Varianten Plusünd Mult", die auf sich selbst verweisen, indem sie jeweils zwei Unterexpressions speichern. Diese rekursive Struktur ermöglicht die Modellierung komplexer mathematischer Ausdrücke. Beachten Sie, dass in den Varianten Plusünd Mult"das Schlüsselwort "Box"verwendet wird. Warum ist dies notwendig? Das liegt daran, dass rekursive Enums in Rust eine feste Größe benötigen, um den Speicherbedarf zur Kompilierzeit zu bestimmen. Durch die Verwendung von "Box"wird ein Zeiger auf den Heap erstellt, der 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 leftünd rightäls "Box¡Exp¿"deklarieren, speichern wir Zeiger auf andere ËxpObjekte im Heap. Dies ermöglicht eine beliebige Verschachtelung von Ausdrücken und ermöglicht es uns, komplexe mathematische Formeln darzustellen. Die Verwendung von "Boxïst daher notwendig, um den rekursiven Bezug in unserem Enum zu realisieren und gleichzeitig eine feste Größe für das Enum-Objekt zur Kompilierzeit zu gewährleisten. Dadurch wird eine effiziente Speicherbelegung erreicht und die Leistung optimiert.

```
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 ËxpEnums in Rust. Das Match-Statement bietet eine effektive Methode, um Enums zu überprüfen und entsprechende Aktionen basierend auf den verschiedenen Varianten auszuführen. Dabei garantiert Rust, dass die Enum-Cases immer wohldefiniert sind, was zu sicherem und zuverlässigem Code führt. Im gegebenen Codebeispiel erstellen wir ein ËxpObjekt namens ë", das die Addition zweier Ganzzahlen repräsentiert. Das ËxpEnum hat drei Varianten: Ïnt", Plusünd Mult". Jede Variante hat 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 Ganzzahlen darstellen. Die Methode ëval"wird auf dem Enum-Objekt aufgerufen, um den Ausdruck auszuwerten. Das Match-Statement wird verwendet, um die verschiedenen Varianten des Enums zu überprüfen und die entsprechenden Aktionen auszuführen. Im vorliegenden Fall wird die PlusVariante erkannt, und das Match-Statement ruft die ëvalMethode für die linken und rechten Unterexpressionen auf und addiert die Ergebnisse. Eine wichtige Eigenschaft des Match-Statements in Rust ist, dass es eine vollständige Überprüfung der möglichen Fälle garantiert. Das bedeutet, dass jede Variante des Enums im Match-Statement abgedeckt sein muss, um sicherzustellen, dass keine Fälle vergessen werden. Dies führt zu sicherem Code, da alle möglichen Fälle behandelt werden und unerwartete Verhaltensweisen vermieden werden. Dank der Garantie von Rust, dass die Enum-Cases immer wohldefiniert 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. Das Match-Statement ist eine leistungsstarke Funktion in Rust, um mit Enums umzugehen und spezifische Aktionen basierend auf den Varianten durchzuführen. Es stellt sicher, dass die Enum-Cases wohldefiniert sind und ermöglicht eine umfassende Kontrolle über den Programmfluss. Im nächsten Kapitel werden wir uns weitere Anwendungsfälle und fortgeschrittene Techniken des Match-Statements ansehen, um sein volles Potenzial auszuschöpfen.

```
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 bestehende ËxpEnum, das die Varianten Ïnt", Plusünd Multënthält. Nun möchten wir eine neue Variante "Div"hinzufügen, die eine Division repräsentiert. Dazu fügen wir den entsprechenden Code zur Enum-Definition hinzu. Nachdem wir die "DivVariante hinzugefügt haben, bleibt der Rest des Codes unverändert. Wir erstellen ein ËxpObjekt ë", das eine Addition zweier Ganzzahlen repräsentiert, und rufen die ëvalMethode auf, um den Ausdruck auszuwerten. Das Match-Statement im ëvalBlock überprüft die verschiedenen Varianten des Enums und führt entsprechende Aktionen aus. Da wir die "DivVariante im Match-Statement nicht abgedeckt haben, wird der Compiler uns einen Fehler melden, da nicht alle möglichen Fälle behandelt sind. Um dieses Problem zu lösen, müssen wir das Match-Statement aktualisieren, um die "DivVariante zu berücksichtigen und ihre Auswertung zu implementieren. Indem wir die neue Variante hinzufügen und ihre logische Auswertung definieren, stellen wir sicher, dass alle Fälle im Match-Statement abgedeckt sind und der Code korrekt funktioniert. Dieses Beispiel zeigt die Stärke und Sicherheit von Rust bei der Arbeit mit Enums. Die Garantie, dass alle Varianten wohldefiniert sind, zwingt uns dazu, alle Fälle im Match-Statement abzudecken und

mögliche unbehandelte Fälle zu vermeiden. 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()
       }
   }
}
```

Beim Versuch den oben gezeigten Code zu kompilieren erhalten wir folgenden Fehler.

```
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 wildca
   |
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

Das ËxpEnum repräsentiert arithmetische Ausdrücke und hat die Varianten Ïnt", Plusünd Mult". Wir haben auch die ëvalMethode implementiert, die den Ausdruck auswertet und das Ergebnis zurückgibt. Im aktuellen Codebeispiel haben wir eine spezielle Anwendung des Match-Statements im ëvalBlock. Innerhalb des Match-Statements für die MultVariante 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 ÏntObjekt mit einem Wert von 0 ist. Wenn dies der Fall ist, wird sofort der Wert 0 zurückgegeben, da das Ergebnis der Multiplikation mit 0 immer 0 ist. Wenn die linke Unterexpression kein ÏntObjekt mit dem Wert 0 ist, wird die Multiplikation der linken und rechten Unterexpressionen durchgeführt und das Ergebnis zurückgegeben. Das Beispiel verdeutlicht das Konzept des

Nested Pattern Matching, bei dem wir innerhalb eines Match-Statements ein weiteres Match-Statement ausführen können, um die verschiedenen Varianten eines Enums weiter zu überprüfen. Durch das Verschachteln von Pattern Matches erhalten wir eine hohe Flexibilität bei der Handhabung komplexer Datenstrukturen. 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, da wir die Logik auf verschiedene Ebenen aufteilen können.

```
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

```
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 spannenden Herausforderung befassen, wie wir die Funktionalität von Rust Enums in der Programmiersprache Java nachbilden können. Rust Enums sind algebraische Datentypen, die eine Vielzahl von Anwendungsmöglichkeiten bieten. Unser Ziel ist es, ähnliche Konzepte und Vorteile in Java zu erreichen. Java verfügt über Enum-Typen, die zur Definition einer festen Menge von Konstanten verwendet werden können. Diese Enums in Java sind jedoch begrenzt und bieten nicht die gleiche Flexibilität wie Rust Enums. Wir werden versuchen, die Funktionalität von Rust Enums in Java nachzubilden, um deren Leistungsfähigkeit und Ausdruckskraft zu erreichen. Eine Möglichkeit, Rust Enums in Java nachzubauen, besteht darin, benutzerdefinierte Klassen zu erstellen, die verschiedene Varianten repräsentieren. Diese Klassen können spezifische Eigenschaften und Methoden haben, um das Verhalten der einzelnen Varianten 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 Variante 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. Indem wir sicherstellen, dass alle Varianten des Enums berücksichtigt werden, können wir eine ähnliche Garantie wie in Rust erhalten. 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. Durch den Nachbau der Funktionalität von Rust Enums in Java können wir die Lei-

stungsfähigkeit und Ausdruckskraft dieser Konstrukte nutzen und die Flexibilität unserer Java-Anwendungen verbessern. Es eröffnet uns neue Möglichkeiten und ermöglicht es uns, komplexe Datenstrukturen und Verhaltensweisen auf elegante und effektive Weise zu modellieren.

2.4.1 Switch Case vs Match

- Veränderung des Enums spielt für SC keine Rolle
- Dieses Verhalten ist auch nicht in Java über tricks Möglich
- Pattern Matching nur über weitere If/Else Abfragen Möglich
- nested pattern matching nur über weitere If Else Möglich

Im Kapitel SSwitch Case vs. Match Statement"vergleichen wir die Verwendung von Switch Case in Java und dem Match Statement in Rust. Hier sind einige wichtige Punkte, auf die wir eingehen werden: 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 das Switch Case in Java einige Einschränkungen: - Die Verwendung von Switch Case ist auf primitive Datentypen wie Integer, Character 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, wenn neue Enum-Werte hinzugefügt werden. Der Compiler warnt nicht vor vergessenen Fällen oder nicht abgedeckten Werten. Im Gegensatz dazu bietet das Match Statement in Rust eine mächtigere und flexiblere Möglichkeit, Pattern Matching durchzuführen. Hier sind einige wichtige Aspekte: - Das Match Statement in Rust kann nicht nur auf Enums, sondern auch auf andere komplexe Datenstrukturen angewendet werden. Es erlaubt das Matching von Mustern, die viel komplexer sind als nur konstante Werte. Dies ermöglicht die Verarbeitung verschiedener Datenstrukturen auf eine deklarative und ausdrucksstarke Weise. - Das Match Statement in Rust garantiert, dass alle möglichen Fälle abgedeckt sind. Wenn ein Fall vergessen wird, gibt der Compiler einen Fehler aus. Dadurch wird vermieden, dass unbehandelte Fälle im Code auftreten und potenzielle Fehler verursachen. - In Java können komplexe Muster und verschachteltes Pattern Matching nur durch weitere If/Else-Abfragen erreicht werden. Dies führt oft zu umständlichem und schwer lesbarerem Code. Im Gegensatz dazu ermöglicht Rust mit dem Match Statement eine klare und gut strukturierte Handhabung von komplexen Mustern und verschachtelten Fällen. Insgesamt bietet das Match Statement in Rust eine leistungsstarke Möglichkeit, Muster und Fälle in Datenstrukturen effizient zu verarbeiten. Es ermöglicht eine sicherere und robustere Code-Entwicklung im Vergleich zum traditionellen Switch Case-Konstrukt in Java.

2.4.2 Expression-Logik in Java

Naiver Ansatz (Geht nicht)

```
public 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();}
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();
    }
}
```

2.4.3 Java Enums am Limit

- Idee, was aber wenn die Instanz ein Wrapper ist
- statische variablen schneiden uns

```
public class EnumLimit{
        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

```
Doggy
Doggy
Catty
new Dog Value
new Dog Value
new Cat value
```

3 Traits

Das Kapitel über Traits behandelt eine zentrale Funktion in Rust, mit der Entwickler Funktionalitäten 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 - ein Typ wird nur akzeptiert, wenn er die Funktionen des Traits implementiert. Ein Trait ist mehr als nur ein Vertrag mit einem Typ; es ist eine Menge von Funktionen oder Verhalten, die über einen Typ verfügbar gemacht werden können. Traits sind Funktionssätze über einem Typen, während Interfaces in der Regel als eigenständige Typen betrachtet werden. Traits in Rust adressieren ähnliche Probleme wie Interfaces in anderen Sprachen, bieten jedoch eine größere Mächtigkeit und Flexibilität. Sie ermöglichen es Entwicklern, gemeinsame Funktionalitäten zu abstrahieren und wiederzuverwenden, indem sie Traits definieren und sie auf verschiedene Typen anwenden. Dadurch wird der Code modularer und flexibler. Zusammenfassend kann gesagt werden, dass Traits in Rust eine leistungsstarke Funktion sind, um geteilte Funktionalitäten zwischen verschiedenen Typen zu realisieren. Sie bieten eine flexiblere Alternative zu herkömmlichen Interfaces und ermöglichen es Entwicklern, abstrakte Konzepte zu modellieren. Durch die Verwendung von Traits können verschiedene Typen über gemeinsame Verhaltensweisen miteinander interagieren und die Wiederverwendbarkeit von Code verbessern.

3.1 Traits in Rust

1. geteilte funktionalität mit anderen Typen
2. Funktionsmenge über einem Typen
3. Oft mit Interfaces verglichen, sind aber keine Interfaces
4. interfaces sind Typen
5. adressieren ähnliche Probleme, traits aber mächtiger

3.1.1 Einfacher Trait

1. Prädikat auf einem Typen

Der Trait SShapefungiert hier als Prädikat für die implementierenden Strukturen. Das bedeutet, dass eine Struktur nur dann als implementierendes Objekt des Traits gilt, wenn sie die Methode äreaïmplementiert. In dem gegebenen Beispiel werden zwei Strukturen, SSquareünd Rectangle", definiert, die beide

den Trait SShapeïmplementieren. Die Methode área"wird jeweils für die Berechnung der Fläche des Quadrats und des Rechtecks implementiert. Durch die Implementierung des Traits SShape"können wir nun auf die Methode áreafür Instanzen von SSquareünd Rectangleßugreifen, unabhängig von der spezifischen Struktur. Dies ermöglicht es uns, die gleiche Funktionalität für verschiedene Formen zu verwenden und den Code zu vereinfachen. Der Trait SShape"dient hier als Prädikat, das sicherstellt, dass eine Struktur über die erforderliche Funktionalität zur Berechnung der Fläche verfügt. Wenn eine Struktur den Trait SShapeïmplementiert, garantieren wir, dass sie über die Methode área"verfügt und somit als gültige geometrische Form betrachtet werden kann. Durch die Verwendung von Traits können wir gemeinsame Verhaltensweisen abstrahieren und wiederverwendbaren Code schreiben. In diesem Beispiel ermöglicht der Trait SShape"die Bereitstellung einer allgemeinen Funktionalität zur Berechnung der Fläche, die von verschiedenen geometrischen Formen verwendet werden kann. Dies verbessert die Modularität und Flexibilität des Codes und ermöglicht es uns, denselben Code für verschiedene Formen zu verwenden, solange sie den Trait SShapeïmplementieren.

```
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

Andere Schreibweise, so kann man die Funktion auf einer Instanz des Structs aufrufen

```
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

1. geht in java auch

```
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{}
```

When running main yields

```
The Animal made a noise
```

3.1.4 Trait Bounds

```
//Das Shape Prädikat muss für A und für B gelten
fn sum_area<A:Shape,B:Shape>(x : &A, y : &B) -> i32 {
    return area(x) + area(y)
 }
```

3.1.5 Multiples Binding

Man kann auch Prädikate/Traits verunden

```
fn sum_area<A:Shape+OtherTraits>(x : &+OtherTraits) -> i32 {
        ...
 }
```

3.1.6 Dynamische Traits

Repräsentieren von Interfaces in Rust
Können Konkrete Typen als Parameter und Rückgabewerte nutzen

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

3.1.7 Kurzschreibweise für dynamische Traits

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

3.1.8 Platzhaltertypen

```
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

```
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>;
}
```

```rust
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

Das Kapitel Supertraits behandelt die Verwendung von Supertraits in Rust. 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 die Anforderungen der vererbten Traits erfüllen muss. In diesem Beispiel wird die Struktur HskaStudent erstellt, die den Traits Person, Student, Programmer und CompSciStudent entspricht. Die Implementierungen der Methoden für die Traits verwenden die Felder der Struktur, um die entsprechenden Werte zurückzugeben. Durch die Verwendung von Supertraits können wir sicherstellen, dass eine Struktur alle Anforderungen erfüllt, die von den vererbten Traits festgelegt werden. Dies ermöglicht eine verbesserte Modularität und Wiederverwendbarkeit von Code, da wir auf vererbte Methoden und Eigenschaften zugreifen können, ohne sie erneut implementieren zu müssen.

```rust
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 gewissen Vorteile gegenüber Interfaces, die ich nun anhand einiger einfacher Beispiele erläutern werden. Man sollte anmerken, dass noch mehr Vorteile gibt, also 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 ambigiös 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 mit Interfaces nicht möglich. Später werden wir noch sehen wie wir auch dieses Problem in Java lösen kann. 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 sin in der Lage mittels Self, den Typen zu referenzieren, für den man ihn 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 wird eine Parser Dependency importieren, bei der wird intern aber möchten, dass sie eine sleep Methode unterstützt geht dies nicht. Bei Traits implementiert man einfach für den fremden Datentyp mittels des impl Blocks den gewünschten Trait.

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

```rust
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 structs zu definieren, falls diese von einem generischen Parameter abhängig sind. Im Beispielcode wird dies mittels der Pair structs illustriert. Dieses stellt die Methode cmp_display, die die Werte des Paares vergleicht und den größeren auf der Kommandozeile ausgibt. Das coole daran ist, dass diese Funktionalit nur zur Verfügung steht, wenn T die Traits Display und PartialOrd zur Verfügung stellt, da wir diese sonst nicht vergleichen und ausgeben könnten. 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.

```rust
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. design pattern[4] helfen uns an dieser stelle. gewisse funktionalitäten wie bspw. konditionelle Implementierung, sind aber gar 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 und falls man die gleiche Methode nochmal implementiert, die aus der Super-Klasse überschreiben muss. Hier nochmal das Beispiel von oben aber in Java. Dies führt zur Compile-Zeit zu einem Error.

```
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. 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();
    }
}
```

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 es 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 gibt es kein Problem. 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. Obwohl dies ein häufig auftretendes Problem ist, gibt es auch hierfür ein einfaches Design Pattern. 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, geschiet der Zugriff nur indirekt über die Wrapper-

Klasse. Im Beispiel stellen wir der Klasse ThirdParty mittels WrapperClass, indirekt die zusätzliche Funktionalit von Sleep zur Verfügung.

```
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 Typens

Dieses Problem wird häufig mittels generischen Interfaces oder Klassen gelöst. Als übergebenen generischen Typ kann man einfach die Klasse selbst übergeben.

```
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 ist dies in Java nicht möglich. Wir könnten nur wie im bereits beschrieben, den generischen Parameter aller Paare beschränken. Wenn wir wie im Beispiel das Hunde-Paar instanziieren möchten bekommen wir einen Fehler zur Compile-Zeit.

```
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;
    }
}
```

compiler 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"));
```

4 Vergleich

4.1 Enums

- Enums kombiniert mit Klassen kann Alle Enums nachbauen

4.2 Traits

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`.