TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 15
Polymorphie

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

Generelle Einteilung

Wir werden die verschiedenen gängigen Arten von Polymorphie im Einzelnen durchgehen und in Beziehung zueinander setzen. Dabei werden wir auch punktuell über Racket und Java hinausgehen und in andere Programmiersprachen hineinschauen.

Generell kann man Konstrukte für Polymorphie in Programmiersprachen in drei Kategorien einteilen, abhängig davon, ob die Konformitätsprüfung beziehungsweise Bindung statisch oder dynamisch ist. Wir haben alle drei schon in Racket und Java gesehen. Was mit statischer und dynamischer Konformitätsprüfung beziehungsweise Bindung gemeint ist, werden wir gleich sehen.

Bisher gesehen

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Racket:**
  - **Funktionen, die auf verschiedenen Typen mit verschieden definierten Operationen arbeiten können**

    Bsp. foldr: (X Y -> Y) Y (list of X) -> Y
- **Java:**
  - **Ableitung von Klassen / Implementierung von Interfaces**
  - **Ad-hoc Polymorphie (Methodenüberladung und implizite Konversion)**
  - **Generics**

Hier sehen Sie eine Übersicht der besagten Konstrukte für Polymorphie, die wir in Racket und Java schon gesehen hatten.

Bisher gesehen

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Racket:**
  - **Funktionen, die auf verschiedenen Typen mit verschieden definierten Operationen arbeiten können**

    Bsp. foldr: (X Y -> Y) Y (list of X) -> Y
- **Java:**
  - **Ableitung von Klassen / Implementierung von Interfaces**
  - **Ad-hoc Polymorphie (Methodenüberladung und implizite Konversion)**
  - **Generics**

Wir haben einige Funktionen in Racket gesehen, die auf verschiedene Typen anwendbar sind, und wir haben solche Funktionen auch selbst implementiert.

Bisher gesehen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Racket:
  - Funktionen, die auf verschiedenen Typen mit verschieden definierten Operationen arbeiten können

    Bsp. foldr: (X Y -> Y) Y (list of X) -> Y
- Java:
  - Ableitung von Klassen / Implementierung von Interfaces
  - Ad-hoc Polymorphie (Methodenüberladung und implizite Konversion)
  - Generics

Die drei klassischen Beispiele kennen wir aus dem Kapitel zu funktionaler Abstraktion: filter, foldr und map. Die Funktion foldr arbeitet auf zwei Typen X und Y, die identisch oder unterschiedlich sein können. Tatsächlich können X und Y völlig beliebig gewählt werden.

Bisher gesehen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Racket:
  - Funktionen, die auf verschiedenen Typen mit verschieden definierten Operationen arbeiten können

    Bsp. foldr: (X Y -> Y) Y (list of X) -> Y
- Java:
  - Ableitung von Klassen / Implementierung von Interfaces
  - Ad-hoc Polymorphie (Methodenüberladung und implizite Konversion)
  - Generics

Wichtig ist halt, dass der erste Parameter auf die beiden Typen X und Y passt. Das heißt konkret, alle in Racket vordefinierten Operationen, die diese als Parameter übergebene Funktion auf ihre beiden eigenen Parameter anwendet, müssen für die Typen X und Y dieser beiden Parameter definiert sein. Zum Beispiel Addition, Größenvergleich und Quadratwurzelberechnung mit sqrt sind vordefinierte Operationen, die nicht für jeden denkbaren Typ X und Y definiert sind.

Bisher gesehen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Racket:
  - Funktionen, die auf verschiedenen Typen mit verschieden definierten Operationen arbeiten können

    Bsp. foldr: (X Y -> Y) Y (list of X) -> Y
- Java:
  - Ableitung von Klassen / Implementierung von Interfaces
  - Ad-hoc Polymorphie (Methodenüberladung und implizite Konversion)
  - Generics

Wir haben mehrere Konzepte von Polymorphie in Java kennen gelernt. Das für objektorientierte Sprachen grundlegende Konzept von Polymorphie ist, dass zwei Typen als Basistyp und Subtyp zueinander ins Verhältnis gesetzt werden können. Polymorphie heißt hier: Hinter der Fassade, die der Basistyp bildet, kann sich ein Objekt von einem beliebigen Subtyp verstecken (natürlich auch vom Basistyp selbst). Der Typ ist in gewissen Grenzen frei wählbar, nämlich Basistyp plus alle direkten und indirekten Subtypen des Basistyps.

Bisher gesehen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Racket:
  - Funktionen, die auf verschiedenen Typen mit verschieden definierten Operationen arbeiten können

    Bsp. foldr: (X Y -> Y) Y (list of X) -> Y
- Java:
  - Ableitung von Klassen / Implementierung von Interfaces
  - Ad-hoc Polymorphie (Methodenüberladung und implizite Konversion)
  - Generics

Man kann darüber streiten, aber es hat sich eingebürgert, Überladung von Subroutinen und implizite Konversionen von einem Datentyp in den anderen ebenfalls als eine Art von Polymorphie anzusehen. Der Name dieser Spielart von Polymorphie drückt schon aus, dass das eine eher primitive Form von Polymorphie ist.

***Nebenbemerkung*: Auf den ersten Blick haben Methodenüberladung und implizite Konversion eigentlich eher wenig miteinander zu tun, so dass diese Subsumierung unter einen gemeinsamen Oberbegriff überraschen mag. Aber in gewisser Weise kann man implizite Konversionen als eine Abwandlung der Überladung von Subroutinen sehen: Jede implizite Konversion kann man ja auffassen als eine Subroutine mit einem Parameter und einem Rückgabewert: Der zu konvertierende Wert ist der aktuale Parameter und der konvertierte Wert ist die Rückgabe. Auf Basis dieser beiden Typen wird die richtige Version der Konversion ausgewählt, so wie bei Methodenüberladung die richtige Version der Methode auf Basis der Anzahl und Typen der Parameter ausgewählt wird.**

Bisher gesehen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Racket:
  - Funktionen, die auf verschiedenen Typen mit verschieden definierten Operationen arbeiten können

    Bsp. foldr: (X Y -> Y) Y (list of X) -> Y
- Java:
  - Ableitung von Klassen / Implementierung von Interfaces
  - Ad-hoc Polymorphie (Methodenüberladung und implizite Konversion)
  - Generics

Auch Generics in Java sind ein Beispiel für Polymorphie. Die Typparameter sind die offen gehaltenen, sprich polymorphen Typen.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Konformitätsprüfung**
  - **zur Kompilierzeit → statische Prüfung**
  - **zur Laufzeit → dynamische Prüfung**
- **Ansteuerung der korrekten Implementation**
  - **zur Kompilierzeit → statische Bindung**
  - **zur Laufzeit → dynamische Bindung**

Zur systematischen Einteilung von Konzepten für Polymorphie schauen wir uns zwei Zeitpunkte an: der Zeitpunkt der Übersetzung und der Zeitpunkt, an dem der Ablauf des Programms an die betrachtete Stelle im Quelltext kommt.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Konformitätsprüfung**
  - **zur Kompilierzeit → statische Prüfung**
  - **zur Laufzeit → dynamische Prüfung**
- **Ansteuerung der korrekten Implementation**
  - **zur Kompilierzeit → statische Bindung**
  - **zur Laufzeit → dynamische Bindung**

Jedes wie auch immer geartete Konzept für Polymorphie enthält diese beiden Aspekte: Zu einem bestimmten Zeitpunkt wird geprüft, ob die einzelnen Objekte und Werte in einem polymorphen Programmkontext tatsächlich konform zu den Anforderungen in diesem Kontext sind, also ob die Objekte und Werte tatsächlich die Funktionalität haben, die in diesem Kontext von ihnen abgefragt wird. Zum selben oder einem späteren Zeitpunkt wird die dazu passende Implementation ausgewählt.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Konformitätsprüfung**
  - **zur Kompilierzeit → statische Prüfung**
  - **zur Laufzeit → dynamische Prüfung**
- **Ansteuerung der korrekten Implementation**
  - **zur Kompilierzeit → statische Bindung**
  - **zur Laufzeit → dynamische Bindung**

Bei beiden Aspekten sprechen wir von *statisch*, wenn er schon beim Übersetzen erledigt wird, und von *dynamisch*, wenn dies erst bei Ausführung des betreffenden Programmteils passiert. Bei der Konformitätsprüfung reden wir also von statischer beziehungsweise dynamischer Prüfung.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Konformitätsprüfung**
  - **zur Kompilierzeit → statische Prüfung**
  - **zur Laufzeit → dynamische Prüfung**
- **Ansteuerung der korrekten Implementation**
  - **zur Kompilierzeit → statische Bindung**
  - **zur Laufzeit → dynamische Bindung**

Wenn die Konformitätsprüfung erfolgreich war, dann muss in jedem Konzept für Polymorphie wie gesagt noch ein Mechanismus vorgesehen sein, der die zu diesen Typen passenden Operationen und Subroutinen auswählt. Hier redet man meist davon, diese Operationen zu *binden*.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Konformitätsprüfung**
  - **zur Kompilierzeit → statische Prüfung**
  - **zur Laufzeit → dynamische Prüfung**
- **Ansteuerung der korrekten Implementation**
  - **zur Kompilierzeit → statische Bindung**
  - **zur Laufzeit → dynamische Bindung**

Was auf dieser Folie nur überblicksweise angedeutet wird, schauen wir uns konkreter auf den nächsten Folien bei den einzelnen Konzepten in Racket und Java an, die wir schon kennen gelernt hatten.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konformitätsprüfung
  - zur Kompilierzeit → statische Prüfung
  - zur Laufzeit → dynamische Prüfung
- Ansteuerung der korrekten Implementation
  - zur Kompilierzeit → statische Bindung
  - zur Laufzeit → dynamische Bindung

Funktionen in Racket:

- **Dynamische Prüfung**
- **Dynamische Bindung**

Fangen wir wieder mit Racket an.

***Erinnerung*: Auch Operatoren wie das Pluszeichen für Addition sind Identifier und bezeichnen Funktionen in Racket-Logik.**

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konformitätsprüfung
  - zur Kompilierzeit → statische Prüfung
  - zur Laufzeit → dynamische Prüfung
- Ansteuerung der korrekten Implementation
  - zur Kompilierzeit → statische Bindung
  - zur Laufzeit → dynamische Bindung

Funktionen in Racket:

- **Dynamische Prüfung**
- **Dynamische Bindung**

Von welchem Typ die Operanden einer Operation wie beispielsweise der Addition sind, wird erst in dem Moment geprüft, wenn die Operation mit diesen Operanden ausgeführt werden soll.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konformitätsprüfung
  - zur Kompilierzeit → statische Prüfung
  - zur Laufzeit → dynamische Prüfung
- Ansteuerung der korrekten Implementation
  - zur Kompilierzeit → statische Bindung
  - zur Laufzeit → dynamische Bindung

Funktionen in Racket:

- **Dynamische Prüfung**
- **Dynamische Bindung**

War die Prüfung erfolgreich, dann wird die entsprechende Operation unmittelbar danach angesteuert und ausgeführt.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konformitätsprüfung
  - zur Kompilierzeit → statische Prüfung
  - zur Laufzeit → dynamische Prüfung
- Ansteuerung der korrekten Implementation
  - zur Kompilierzeit → statische Bindung
  - zur Laufzeit → dynamische Bindung

Java – Klassen ableiten / Interfaces implementieren:

- **Statische Prüfung**
- **Dynamische Bindung**

Gehen wir zu Java über, zuerst das Thema Basistyp und Subtypen.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konformitätsprüfung
  - zur Kompilierzeit → statische Prüfung
  - zur Laufzeit → dynamische Prüfung
- Ansteuerung der korrekten Implementation
  - zur Kompilierzeit → statische Bindung
  - zur Laufzeit → dynamische Bindung

Java – Klassen ableiten / Interfaces implementieren:

- **Statische Prüfung**
- **Dynamische Bindung**

Zu prüfen ist, ob der dynamische Typ einer Referenz gleich ihrem statischen Typ beziehungsweise ein Subtyp ihres statischen Typs ist (oder Wert null hat). Weiter zu prüfen ist, dass nur Funktionalität verwendet wird, die schon im Basistyp definiert ist. Diese Prüfungen können schon bei der Übersetzung geleistet werden, weil Inspektion des Quelltextes dafür reicht. Wie Sie wissen, wird das in Java auch so gemacht.

***Erinnerung*: Kapitel 03b, Abschnitt zu Subtypen sowie statischem und dynamischem Typ.**

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konformitätsprüfung
  - zur Kompilierzeit → statische Prüfung
  - zur Laufzeit → dynamische Prüfung
- Ansteuerung der korrekten Implementation
  - zur Kompilierzeit → statische Bindung
  - zur Laufzeit → dynamische Bindung

Java – Klassen ableiten / Interfaces implementieren:

- **Statische Prüfung**
- **Dynamische Bindung**

Aber wir haben auch gesehen, dass die auszuführende Implementation der aufgerufenen Methode erst direkt beim Aufruf der Methode ausgewählt und angesteuert wird, die Bindung ist also dynamisch.

***Erinnerung*: Kapitel 03c, Abschnitt zu verborgenen Informationen.**

***Nebenbemerkung*: In vielen recht einfachen Fällen könnte ein Compiler prinzipiell den dynamischen Typ eines Objektes identifizieren und könnte die Methodenaufrufe auf diesem Objekt schon statisch binden – aber eben nicht generell.**

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konformitätsprüfung
  - zur Kompilierzeit → statische Prüfung
  - zur Laufzeit → dynamische Prüfung
- Ansteuerung der korrekten Implementation
  - zur Kompilierzeit → statische Bindung
  - zur Laufzeit → dynamische Bindung

Ad-hoc Polymorphie:

- **Statische Prüfung**
- **Bindung statisch oder dynamisch?**

Bei Ad-hoc Polymorphie, also bei Methodenüberladung und impliziter Konversion, sieht die Sache anders aus.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konformitätsprüfung
  - zur Kompilierzeit → statische Prüfung
  - zur Laufzeit → dynamische Prüfung
- Ansteuerung der korrekten Implementation
  - zur Kompilierzeit → statische Bindung
  - zur Laufzeit → dynamische Bindung

Ad-hoc Polymorphie:

- **Statische Prüfung**
- **Bindung statisch oder dynamisch?**

Zu prüfen bei Methodenüberladung ist, ob in jeder Klasse alle Methoden unterschiedliche Signaturen haben. Weiter zu prüfen ist, ob jeder Methodenaufruf unzweideutig genau einer Methodendefinition zugeordnet werden kann.

Bei impliziten Konversionen ist zu prüfen, ob der zu konvertierende Wert tatsächlich von einem Typ ist, der implizit in den Zieltyp konvertiert werden kann.

Auch all dies kann schon bei der Übersetzung erledigt werden und wird in Java auch schon zu diesem Zeitpunkt erledigt.

***Erinnerung*: für Methodenüberladung Kapitel 03c, Abschnitt zur Signatur und zum Überschreiben und Überladen von Methoden; für Konversionen der gleichnamige Abschnitt in Kapitel 01b.**

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konformitätsprüfung
  - zur Kompilierzeit → statische Prüfung
  - zur Laufzeit → dynamische Prüfung
- Ansteuerung der korrekten Implementation
  - zur Kompilierzeit → statische Bindung
  - zur Laufzeit → dynamische Bindung

Ad-hoc Polymorphie:

- **Statische Prüfung**
- **Bindung statisch oder dynamisch?**

Das Thema Bindung ist etwas diffiziler. Da müssen wir auch zwischen Methodenüberladung und impliziten Konversionen unterscheiden.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konformitätsprüfung
  - zur Kompilierzeit → statische Prüfung
  - zur Laufzeit → dynamische Prüfung
- Ansteuerung der korrekten Implementation
  - zur Kompilierzeit → statische Bindung
  - zur Laufzeit → dynamische Bindung

Implizite Konversionen:

- **Statische Prüfung**
- **Statische Bindung**

Bei impliziten Konversionen ist die Sachlage einfach: Die jeweils passende implizite Konversion wird schon bei der Übersetzung eingesetzt, also statische Bindung.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konformitätsprüfung
  - zur Kompilierzeit → statische Prüfung
  - zur Laufzeit → dynamische Prüfung
- Ansteuerung der korrekten Implementation
  - zur Kompilierzeit → statische Bindung
  - zur Laufzeit → dynamische Bindung

Methodenüberladung:

- **Statische Prüfung**
- **Statische Bindung wäre *eigentlich* möglich**

Generell wird in Java die anzusteuernde Implementation einer Methode erst zur Laufzeit ausgewählt, unmittelbar vor Ausführung des Aufrufs selbst. Daher ist die Bindung zwangsläufig dynamisch. Warum steht da aber etwas anderes auf der Folie?

Nun, das Konzept Methodenüberladung selbst erzwingt eigentlich nicht unbedingt dynamische Bindung. Wäre dynamische Bindung nicht von vornherein in Java eingebaut, dann wäre statische Bindung hier möglich, das heißt, die korrekte Variante der aufgerufenen Methode könnte schon beim Übersetzen ausgewählt und eingesetzt werden.

***Nebenbemerkung*: Zum Beispiel in C++ kann man bei der Definition von Klassen und Methoden entscheiden, ob Methoden statisch oder dynamisch gebunden werden. Im ersteren Fall steht der statischen Bindung bei Methodenüberladung nichts im Wege.**

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konformitätsprüfung
  - zur Kompilierzeit → statische Prüfung
  - zur Laufzeit → dynamische Prüfung
- Ansteuerung der korrekten Implementation
  - zur Kompilierzeit → statische Bindung
  - zur Laufzeit → dynamische Bindung

Java – Generics:

- **Statische Prüfung**
- **Statische Bindung wäre *eigentlich* möglich**

Der letzte Punkt auf unserer Liste von schon gesehenen Konzepten für Polymorphie sind Generics in Java.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konformitätsprüfung
  - zur Kompilierzeit → statische Prüfung
  - zur Laufzeit → dynamische Prüfung
- Ansteuerung der korrekten Implementation
  - zur Kompilierzeit → statische Bindung
  - zur Laufzeit → dynamische Bindung

Java – Generics:

- **Statische Prüfung**
- **Statische Bindung wäre *eigentlich* möglich**

Schon bei der Übersetzung wird geprüft, ob jeder Typparameter die Funktionalität hat, die ihm in der generischen Klasse beziehungsweise Methode abverlangt wird.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Konformitätsprüfung
  - zur Kompilierzeit → statische Prüfung
  - zur Laufzeit → dynamische Prüfung
- Ansteuerung der korrekten Implementation
  - zur Kompilierzeit → statische Bindung
  - zur Laufzeit → dynamische Bindung

Java – Generics:

- **Statische Prüfung**
- **Statische Bindung wäre *eigentlich* möglich**

Aus denselben Gründen wie eben bei Metodenüberladung verhält sich das Thema Bindung genau so wie eben bei der Methodenüberladung.

***Vorgriff*: Wir werden in Kürze Konzepte aus zwei anderen Programmiersprachen sehen, die analog zu Generics in Java sind, aber in diesen Sprachen ist dynamische Bindung nicht von vornherein vorgesehen, so dass der Compiler tatsächlich schon bei der Übersetzung die richtigen Subroutinen einsetzt.**

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Racket: dynamische Bindung und Prüfung
- Java – Klassen ableiten / Interfaces implementieren: statische Prüfung, dynamische Bindung
- Java – Generics: statische Prüfung und prinzipiell auch statische Bindung

Vierter Fall: dynamische Prüfung, statische Bindung

- Macht keinen Sinn
- Nach KWs Kenntnis in keiner Sprache realisiert

Auf den letzten Folien haben wir drei Kombinationen gesehen: statische Prüfung mit statischer Bindung, statische Prüfung mit dynamischer Bindung und dynamische Prüfung mit dynamischer Bindung. Oben sind diese drei Fälle der Vollständigkeit halber noch einmal aufgeführt.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Racket: dynamische Bindung und Prüfung
- Java – Klassen ableiten / Interfaces implementieren: statische Prüfung, dynamische Bindung
- Java – Generics: statische Prüfung und prinzipiell auch statische Bindung

Vierter Fall: dynamische Prüfung, statische Bindung

- Macht keinen Sinn
- Nach KWs Kenntnis in keiner Sprache realisiert

Was ist nun mit dem vierten Fall: dass die Prüfung erst bei der Ausführung, die Bindung aber schon bei der Übersetzung passiert?

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Racket: dynamische Bindung und Prüfung
- Java – Klassen ableiten / Interfaces implementieren: statische Prüfung, dynamische Bindung
- Java – Generics: statische Prüfung und prinzipiell auch statische Bindung

Vierter Fall: dynamische Prüfung, statische Bindung

- Macht keinen Sinn
- Nach KWs Kenntnis in keiner Sprache realisiert

Nun, die Bindung folgt ja zwangsläufig nach der Prüfung. Daher ist dieser vierte Fall logisch in sich widersprüchlich.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

Typische Begriffsverwendung:

- **Duck-Typing: dynamische Prüfung und Bindung**
- **Subtyppolymorphie: statische Prüfung und dynamische Bindung wie in Java über Subtypen**
- **Statische Prüfung und Bindung: Generizität**

Caveat:

- **Begriffsverwendung nicht 100% einheitlich!**
- **Zu Generizität siehe Problematik letzte Folien.**

Zum Ende dieses Abschnitts möchte ich Ihnen die gängigen Bezeichnungen für die drei identifizierten Fälle nicht vorenthalten.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

Typische Begriffsverwendung:

- **Duck-Typing: dynamische Prüfung und Bindung**
- **Subtyppolymorphie: statische Prüfung und dynamische Bindung wie in Java über Subtypen**
- **Statische Prüfung und Bindung: Generizität**

Caveat:

- **Begriffsverwendung nicht 100% einheitlich!**
- **Zu Generizität siehe Problematik letzte Folien.**

Woher dieser Name kommt, ist wohl nicht hundertprozentig geklärt. Die Wikipedia spekuliert, dass der Begriff Duck Test vielleicht Pate stand.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

Typische Begriffsverwendung:

- **Duck-Typing: dynamische Prüfung und Bindung**
- **Subtyppolymorphie: statische Prüfung und dynamische Bindung wie in Java über Subtypen**
- **Statische Prüfung und Bindung: Generizität**

Caveat:

- **Begriffsverwendung nicht 100% einheitlich!**
- **Zu Generizität siehe Problematik letzte Folien.**

Zum polymorphen Verhältnis zwischen Basistyp und Subtyp, wie Sie es aus Java kennen, finden Sie weitere Quellen unter dem Begriff Subtyppolymorphie.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

Typische Begriffsverwendung:

- **Duck-Typing: dynamische Prüfung und Bindung**
- **Subtyppolymorphie: statische Prüfung und dynamische Bindung wie in Java über Subtypen**
- **Statische Prüfung und Bindung: Generizität**

Caveat:

- **Begriffsverwendung nicht 100% einheitlich!**
- **Zu Generizität siehe Problematik letzte Folien.**

Hier sehen wir, dass der Begriff Generics in Java nicht zufällig gewählt wurde: Generics sind eben das generische Konzept in Java. In vielen – nicht allen – Programmiersprachen, die Generizität enthalten, heißt das entsprechende Programmierkonzept ebenfalls Generics.

Generelle Einteilung

TECHNISCHE UNIVERSITÄT DARMSTADT

Typische Begriffsverwendung:

- **Duck-Typing: dynamische Prüfung und Bindung**
- **Subtyppolymorphie: statische Prüfung und dynamische Bindung wie in Java über Subtypen**
- **Statische Prüfung und Bindung: Generizität**

Caveat:

- **Begriffsverwendung nicht 100% einheitlich!**
- **Zu Generizität siehe Problematik letzte Folien.**

Wir hatten schon gesehen, dass diese Definition von Generizität nicht hundertprozentig dem allgemeinen Sprachgebrauch entspricht. Aber der Unterschied kommt eben daher, dass Subtyppolymorphie dynamische Bindung hat und diese dynamische Bindung in manchen Sprachen wie Java immer da ist, egal ob man Subtyppolymorphie überhaupt einsetzen will oder nicht. Generics allein für sich erfordern keine dynamische Bindung und sind in anderen Sprachen auch mit statischer Bindung realisiert.

TECHNISCHE UNIVERSITÄT DARMSTADT

Abstraktion und Polymorphie

Im Grunde kann man es so ausdrücken: Polymorphie ist die Möglichkeit, Abstraktion auf Typebene in Programmiersprachen auszudrücken. Um zu verstehen, was Polymorphie eigentlich ist, müssen wir also zwei Dinge klären: was wir hier unter dem etwas unscharfen Begriff Abstraktion genau zu verstehen haben und was es mit der Typebene auf sich hat.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

***Abstraktion*: einen „Wust“ von Entitäten gedanklich in geeigneter Form strukturieren**

- **Zuordnung der Entitäten zu passenden Kategorien**
  - **Zuweilen etwas „mit Gewalt“**
- **Bei jeder Kategorie**
  - **das Gemeinsame aller Elemente einer Kategorie (ggf. in mehrere Aspekte) herausfaktorisieren und**
  - **das Unterschiedliche jeweils für sich stehen lassen**
- **Beziehungen zwischen den Kategorien herstellen**

Klären wir also erst einmal den Begriff Abstraktion. Die Erläuterung auf dieser Folie sollte eigentlich allgemeingültig sein, auch jenseits der Informatik.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

***Abstraktion*: einen „Wust“ von Entitäten gedanklich in geeigneter Form strukturieren**

- **Zuordnung der Entitäten zu passenden Kategorien**
  - **Zuweilen etwas „mit Gewalt“**
- **Bei jeder Kategorie**
  - **das Gemeinsame aller Elemente einer Kategorie (ggf. in mehrere Aspekte) herausfaktorisieren und**
  - **das Unterschiedliche jeweils für sich stehen lassen**
- **Beziehungen zwischen den Kategorien herstellen**

Wir sprechen bei Abstraktion immer von Mengen, auf denen der Abstraktionsprozess geschieht. Der Begriff Entität lässt völlig offen, was für eine Art von Elementen in einer Menge ist.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

***Abstraktion*: einen „Wust“ von Entitäten gedanklich in geeigneter Form strukturieren**

- **Zuordnung der Entitäten zu passenden Kategorien**
  - **Zuweilen etwas „mit Gewalt“**
- **Bei jeder Kategorie**
  - **das Gemeinsame aller Elemente einer Kategorie (ggf. in mehrere Aspekte) herausfaktorisieren und**
  - **das Unterschiedliche jeweils für sich stehen lassen**
- **Beziehungen zwischen den Kategorien herstellen**

An diesen Punkt wird man sicher schnell denken: Abstraktion ist Generalisierung ...

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

***Abstraktion*: einen „Wust“ von Entitäten gedanklich in geeigneter Form strukturieren**

- **Zuordnung der Entitäten zu passenden Kategorien**
  - **Zuweilen etwas „mit Gewalt“**
- **Bei jeder Kategorie**
  - **das Gemeinsame aller Elemente einer Kategorie (ggf. in mehrere Aspekte) herausfaktorisieren und**
  - **das Unterschiedliche jeweils für sich stehen lassen**
- **Beziehungen zwischen den Kategorien herstellen**

... und das, was bei den einzelnen Elementen nach der Generalisierung stehen bleibt, sind eher untergeordnete Details.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

***Abstraktion*: einen „Wust“ von Entitäten gedanklich in geeigneter Form strukturieren**

- **Zuordnung der Entitäten zu passenden Kategorien**
  - **Zuweilen etwas „mit Gewalt“**
- **Bei jeder Kategorie**
  - **das Gemeinsame aller Elemente einer Kategorie (ggf. in mehrere Aspekte) herausfaktorisieren und**
  - **das Unterschiedliche jeweils für sich stehen lassen**
- **Beziehungen zwischen den Kategorien herstellen**

Aber Abstraktion bedeutet nicht nur *Generalisierung*, sondern gleichermaßen auch *Differenzierung*.

Ein Beispiel aus der Welt außerhalb der Informatik sind die beiden Abstraktionen „Bio“ und „Öko“. Häufig werden diese beiden Begriffe miteinander vermischt, aber eigentlich gibt es eine ganz klare Unterscheidung: „ökologisch“ bezieht sich auf umweltschonende und artgerechte Produktion; biologisch bezieht auch die Gesundheit der Konsumenten mit ein.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

***Abstraktion*: einen „Wust“ von Entitäten gedanklich in geeigneter Form strukturieren**

- **Zuordnung der Entitäten zu passenden Kategorien**
  - **Zuweilen etwas „mit Gewalt“**
- **Bei jeder Kategorie**
  - **das Gemeinsame aller Elemente einer Kategorie (ggf. in mehrere Aspekte) herausfaktorisieren und**
  - **das Unterschiedliche jeweils für sich stehen lassen**
- **Beziehungen zwischen den Kategorien herstellen**

Ob der Begriff „biologisch“ den Begriff „ökologisch“ umfasst oder sich *nur* auf die Gesundheit der Konsumenten bezieht und damit klar von „ökologisch“ abgegrenzt ist, da scheint die Begriffsverwendung nicht einheitlich zu sein. Aber um vernünftig handhabbare Begriffe zu haben, etwa bei gesetzlichen Regelungen, wird man oft nicht umhin können, etwas rabiat zu sein und sich *entweder* für das Eine *oder* für das Andere zu entscheiden.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

***Abstraktion*: einen „Wust“ von Entitäten gedanklich in geeigneter Form strukturieren**

- **Zuordnung der Entitäten zu passenden Kategorien**
  - **Zuweilen etwas „mit Gewalt“**
- **Bei jeder Kategorie**
  - **das Gemeinsame aller Elemente einer Kategorie (ggf. in mehrere Aspekte) herausfaktorisieren und**
  - **das Unterschiedliche jeweils für sich stehen lassen**
- **Beziehungen zwischen den Kategorien herstellen**

Das ist dann auch ein Beispiel für den letzten Schritt im Abstraktionsprozess: Entweder stehen „ökologisch“ und „biologisch“ gleichsam als Zwillinge nebeneinander, oder „biologisch“ ist ein Überbegriff für „ökologisch“ in dem Sinn, dass nur das, was *öko*logisch ist, auch *bio*logisch sein kann.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

***Abstraktion*: einen „Wust“ von Entitäten gedanklich in geeigneter Form strukturieren**

- **Zuordnung der Entitäten zu passenden Kategorien**
  - **Zuweilen etwas „mit Gewalt“**
- **Bei jeder Kategorie**
  - **das Gemeinsame aller Elemente einer Kategorie (ggf. in mehrere Aspekte) herausfaktorisieren und**
  - **das Unterschiedliche jeweils für sich stehen lassen**
- **Beziehungen zwischen den Kategorien herstellen**

Die gemeinsame Abstraktion kann aus mehreren, verschiedenen Aspekten bestehen. Häufig ist es sinnvoll, diese Aspekte auseinanderzudividieren und diese dann hinterher wieder zu koppeln.

***Vorgriff*: Wir werden gleich sehen, dass genau dies bei den GUI-Komponenten in AWT beziehungsweise Swing gemacht wurde.**

***Erinnerung*: AWT und Swing hatten wir in entsprechenden Abschnitten von Kapitel 10 behandelt.**

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Racket: fold, filter, map

- **Entitäten: Datenverarbeitungsaufgaben mittels einmaligem Durchlauf einer Liste**
- **Kategorie: fold, filter, map**
- **Bei jeder Kategorie**
  - **Gemeinsamkeit in die jeweilige Funktion (fold, filter oder map) herausfaktorisiert**
  - **Unterschiede: Parameter neben der Liste**
- **Beziehungen: filter und map sind *intermediate***

Das grundlegende, immer wieder zitierte Beispiel aus funktionaler Abstraktion haben wir im gleichnamigen Kapitel ebenfalls behandelt: die allgemeinen Funktionen fold, filter und map.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Racket: fold, filter, map

- **Entitäten: Datenverarbeitungsaufgaben mittels einmaligem Durchlauf einer Liste**
- **Kategorie: fold, filter, map**
- **Bei jeder Kategorie**
  - **Gemeinsamkeit in die jeweilige Funktion (fold, filter oder map) herausfaktorisiert**
  - **Unterschiede: Parameter neben der Liste**
- **Beziehungen: filter und map sind *intermediate***

Vor langer Zeit ist den Informatikern aufgefallen, dass sie – abstrakt betrachtet – immer und immer wieder dieselben Standardaufgaben zu lösen hatten, jeweils in unterschiedlichen Anwendungskontexten mit unterschiedlichen Datentypen und unterschiedlichen Funktionsparametern.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Racket: fold, filter, map

- **Entitäten: Datenverarbeitungsaufgaben mittels einmaligem Durchlauf einer Liste**
- **Kategorie: fold, filter, map**
- **Bei jeder Kategorie**
  - **Gemeinsamkeit in die jeweilige Funktion (fold, filter oder map) herausfaktorisiert**
  - **Unterschiede: Parameter neben der Liste**
- **Beziehungen: filter und map sind *intermediate***

Man hat erkannt, dass es vorwiegend drei Arten von Datenverarbeitungsaufgaben waren, nämlich die drei, die Sie unter den Namen fold, filter und map kennen.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Racket: fold, filter, map

- **Entitäten: Datenverarbeitungsaufgaben mittels einmaligem Durchlauf einer Liste**
- **Kategorie: fold, filter, map**
- **Bei jeder Kategorie**
  - **Gemeinsamkeit in die jeweilige Funktion (fold, filter oder map) herausfaktorisiert**
  - **Unterschiede: Parameter neben der Liste**
- **Beziehungen: filter und map sind *intermediate***

Die drei Funktionen fold, filter und map, wie wir sie in Kapitel 04c gesehen haben, bilden genau die herausfaktorisierten Gemeinsamkeiten.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Racket: fold, filter, map

- **Entitäten: Datenverarbeitungsaufgaben mittels einmaligem Durchlauf einer Liste**
- **Kategorie: fold, filter, map**
- **Bei jeder Kategorie**
  - **Gemeinsamkeit in die jeweilige Funktion (fold, filter oder map) herausfaktorisiert**
  - **Unterschiede: Parameter neben der Liste**
- **Beziehungen: filter und map sind *intermediate***

Was macht eine konkrete Ausprägung eines der drei abstrakten Konzepte – fold, filter und map – nun aus? Die Antwort auf diese Frage liefert die jeweilige Parameterliste. Neben der Eingabeliste haben alle drei Funktionen jeweils eine Funktion als Parameter, fold hat zudem noch einen Wert vom Ergebnistyp als weiteren Parameter. In den verschiedenen Anwendungen einer dieser drei Funktionen variieren diese Parameter. Das sind die übriggebliebenen Unterschiede zwischen den einzelnen Entitäten, die wir eben untergeordnete Details genannt hatten.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Racket: fold, filter, map

- **Entitäten: Datenverarbeitungsaufgaben mittels einmaligem Durchlauf einer Liste**
- **Kategorie: fold, filter, map**
- **Bei jeder Kategorie**
  - **Gemeinsamkeit in die jeweilige Funktion (fold, filter oder map) herausfaktorisiert**
  - **Unterschiede: Parameter neben der Liste**
- **Beziehungen: filter und map sind *intermediate***

Beziehungen zwischen den Kategorien können von unterschiedlichster Art sein. In diesem Fall hatten wir schon diese Beziehung in Kapitel 08 festgestellt: Die Operationen filter und map sind intermediate, die Operation fold ist terminal.

Man könnte also daran denken, filter und map nicht so stehenzulassen, sondern sie zu einer Funktion zu vereinigen, die intermediate operations verschiedenster Art realisieren würde. Das ginge schon. Aber da fold einen Parameter mehr hat als map. wäre die Mühe bei der Verwendung einer solchen Funktion wahrscheinlich höher als der Gewinn, wenn man anstelle von filter und map nur noch eine gemeinsame Funktion hätte.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Java AWT / Swing:

- *Component*: Button, Canvas, Checkbox, Choice, Label, List, Scrollbar, TextArea, TextField
- *Listener*: KeyListener, MouseListener, MouseMotionListener, MouseWheelListener, WindowListener, WindowFocusListener, WindowStateListener
- *Event*: ActionEvent, KeyEvent, MouseEvent, MouseWheelEvent, WindowEvent

Jetzt ein weiteres, größeres Beispiel, nunmehr in Java. Was wir bisher zum Thema Abstraktion gesagt haben, hat Sie vielleicht an das Thema GUIs erinnert – zu Recht.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Java AWT / Swing:

- *Component*: Button, Canvas, Checkbox, Choice, Label, List, Scrollbar, TextArea, TextField
- *Listener*: KeyListener, MouseListener, MouseMotionListener, MouseWheelListener, WindowListener, WindowFocusListener, WindowStateListener
- *Event*: ActionEvent, KeyEvent, MouseEvent, MouseWheelEvent, WindowEvent

Das erste, offensichtliche Beispiel sind die Arten von GUI-Komponenten. Die gemeinsame Abstraktion ist eben, dass dies alles Arten von Komponenten in einem GUI sind. Die Gemeinsamkeiten sind in die gemeinsame Basisklasse Component herausfaktorisiert. Dadurch kann man beispielsweise GUI-Komponenten verschiedener Arten unterschiedslos in einem Container nebeneinander speichern und völlig gleich behandeln, ohne wissen zu müssen, von welchem Typ die einzelne Komponente im Container ist.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Java AWT / Swing:

- ***Component*: Button, Canvas, Checkbox, Choice, Label, List, Scrollbar, TextArea, TextField**
- ***Listener*: KeyListener, MouseListener, MouseMotionListener, MouseWheelListener, WindowListener, WindowFocusListener, WindowStateListener**
- ***Event*: ActionEvent, KeyEvent, MouseEvent, MouseWheelEvent, WindowEvent**

Sie erinnern sich: Diese beiden Klassen waren nicht direkt, sondern indirekt von Klasse Component abgeleitet worden, und zwar über Klasse TextComponent, die für sich genommen eigentlich keine große Rolle spielt. Offensichtlich haben TextArea und TextField viele Gemeinsamkeiten und wenig Unterschiede, so dass man sich entschieden hat, die Gemeinsamkeiten in eine Klasse TextComponent herauszufaktorisieren.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Java AWT / Swing:

- ***Component*: Button, Canvas, Checkbox, Choice, Label, List, Scrollbar, TextArea, TextField**
- ***Listener*: KeyListener, MouseListener, MouseMotionListener, MouseWheelListener, WindowListener, WindowFocusListener, WindowStateListener**
- ***Event*: ActionEvent, KeyEvent, MouseEvent, MouseWheelEvent, WindowEvent**

Eigentlich gehört die Funktionalität der Listener zu den einzelnen GUI-Komponenten. Man hat sich aber dafür entschieden, diese Funktionalität aus Klasse Component und den von Component abgeleiteten Klassen herauszufaktorisieren.

***Erinnerung*: Gerade eben hatten wir vorangekündigt, dass AWT und Swing ein Beispiel dafür sind, dass die Gemeinsamkeiten nicht immer nur in eine einzelne Abstraktion herausfaktorisiert werden, sondern in mehrere Aspekte, die dann hinterher wieder miteinander verknüpft werden. Die Verknüpfung geschieht hier über die Registrierung mit Methoden wie addKeyListener und so weiter.**

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Java AWT / Swing:

- *Component*: Button, Canvas, Checkbox, Choice, Label, List, Scrollbar, TextArea, TextField
- *Listener*: KeyListener, MouseListener, MouseMotionListener, MouseWheelListener, WindowListener, WindowFocusListener, WindowStateListener
- *Event*: ActionEvent, KeyEvent, MouseEvent, MouseWheelEvent, WindowEvent

Die Informationen zum Event wurden dann nochmals aus dem Konzept Listener ausgelagert in das Konzept Event-Klassen. Diese kommen dann als Parameter der Methoden in die Listener zurück.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Abstraktion auf Typebene:

- **In einer logischen Einheit (Klasse, Subroutine...) sind ein oder mehrere Typen nicht festgelegt.**
- **Können bei der Nutzung der Einheit aus einer Menge von Typen gewählt werden.**
- **Gewählte Typen müssen die Funktionalität bieten, die von ihnen in der Einheit abverlangt wird.**
- **Bei mehreren offenen Typen muss auch die gemeinsame Funktionalität korrekt vorhanden sein.**
  - **Z.B. bei foldr (X Y -> Y) mit den korrekten Typen X und Y.**

Was bedeutet das jetzt ganz allgemein gesprochen?

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Abstraktion auf Typebene:

- **In einer logischen Einheit (Klasse, Subroutine...) sind ein oder mehrere Typen nicht festgelegt.**
- **Können bei der Nutzung der Einheit aus einer Menge von Typen gewählt werden.**
- **Gewählte Typen müssen die Funktionalität bieten, die von ihnen in der Einheit abverlangt wird.**
- **Bei mehreren offenen Typen muss auch die gemeinsame Funktionalität korrekt vorhanden sein.**
  - **Z.B. bei foldr (X Y -> Y) mit den korrekten Typen X und Y.**

Das ist sicherlich die Gemeinsamkeit aller Beispiele für Polymorphie in Racket und Java, auf die wir auf den letzten Folien noch einmal zurückgeschaut hatten. Typen sind nicht festgelegt, sondern in Grenzen polymorph ...

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Abstraktion auf Typebene:

- **In einer logischen Einheit (Klasse, Subroutine...) sind ein oder mehrere Typen nicht festgelegt.**
- **Können bei der Nutzung der Einheit aus einer Menge von Typen gewählt werden.**
- **Gewählte Typen müssen die Funktionalität bieten, die von ihnen in der Einheit abverlangt wird.**
- **Bei mehreren offenen Typen muss auch die gemeinsame Funktionalität korrekt vorhanden sein.**
  - **Z.B. bei foldr (X Y -> Y) mit den korrekten Typen X und Y.**

... und werden erst bei Verwendung festgelegt.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Abstraktion auf Typebene:

- **In einer logischen Einheit (Klasse, Subroutine...) sind ein oder mehrere Typen nicht festgelegt.**
- **Können bei der Nutzung der Einheit aus einer Menge von Typen gewählt werden.**
- **Gewählte Typen müssen die Funktionalität bieten, die von ihnen in der Einheit abverlangt wird.**
- **Bei mehreren offenen Typen muss auch die gemeinsame Funktionalität korrekt vorhanden sein.**
  - **Z.B. bei foldr (X Y -> Y) mit den korrekten Typen X und Y.**

Natürlich muss der gewählte Typ auch in den Kontext passen. Das wird durch die Konformitätsprüfung gewährleistet.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Abstraktion auf Typebene:

- **In einer logischen Einheit (Klasse, Subroutine...) sind ein oder mehrere Typen nicht festgelegt.**
- **Können bei der Nutzung der Einheit aus einer Menge von Typen gewählt werden.**
- **Gewählte Typen müssen die Funktionalität bieten, die von ihnen in der Einheit abverlangt wird.**
- **Bei mehreren offenen Typen muss auch die gemeinsame Funktionalität korrekt vorhanden sein.**
  - **Z.B. bei foldr (X Y -> Y) mit den korrekten Typen X und Y.**

Wir hatten es bisher nicht explizit thematisiert, aber wenn mehrere polymorphe Typen miteinander interagieren sollen, dann müssen sie auch zusammenpassen.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Abstraktion auf Typebene:

- **In einer logischen Einheit (Klasse, Subroutine...) sind ein oder mehrere Typen nicht festgelegt.**
- **Können bei der Nutzung der Einheit aus einer Menge von Typen gewählt werden.**
- **Gewählte Typen müssen die Funktionalität bieten, die von ihnen in der Einheit abverlangt wird.**
- **Bei mehreren offenen Typen muss auch die gemeinsame Funktionalität korrekt vorhanden sein.**
  - **Z.B. bei foldr (X Y -> Y) mit den korrekten Typen X und Y.**

Zum Beispiel hier müssen die offen gehaltenen Typen X und Y miteinander interagieren. Diese Interaktion ist ausgelagert in den Funktionsparameter. Die an foldr übergebene Funktion muss diese Interaktion korrekt realisieren.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

***Polymorphie*:**

Oberbegriff für alle programmiersprachlichen Konzepte (wie die in Racket und Java gesehenen), mit denen Abstraktion auf Typebene realisiert werden kann.

Jetzt sind wir soweit, den Begriff Polymorphie zu definieren, indem wir ihn auf die Begriffe Abstraktion und Typebene zurückführen, die wir soeben geklärt hatten.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Wozu Polymorphie: Separation of Concerns

- **Möglichst viele Typen gleichbehandeln an Stellen wo die Unterschiede zwischen den Typen nicht wichtig sind.**
- **Jede Einheit (Klasse, Subroutine...), deren Logik auf verschiedene Typen passt, möglichst nur einmal implementieren.**
  - **Die Unterschiede dann durch Füllen der offen gebliebenen Lücke.**
- **Beispiele:**

  foldr: (X Y -> Y) Y (list of X) -> Y

  Collections.sort (List<T>, Comparator<T>)

Die Anschlussfrage ist, wozu Polymorphie gut ist. Es muss ja gute Gründe geben, wenn Polymorphie in so vielen Programmiersprachen realisiert ist, teilweise sogar mit mehreren unterschiedlichen Konzepten in derselben Programmiersprache.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Wozu Polymorphie: Separation of Concerns

- **Möglichst viele Typen gleichbehandeln an Stellen wo die Unterschiede zwischen den Typen nicht wichtig sind.**
- **Jede Einheit (Klasse, Subroutine...), deren Logik auf verschiedene Typen passt, möglichst nur einmal implementieren.**
  - **Die Unterschiede dann durch Füllen der offen gebliebenen Lücke.**
- **Beispiele:**

  foldr: (X Y -> Y) Y (list of X) -> Y

  Collections.sort (List<T>, Comparator<T>)

Die Antwort kennen wir aus Kapitel 14, Stichwort Separation of Concerns. Wobei Separations of Concern natürlich über Polymorphie hinausgeht. Beispielsweise MVC basiert nicht zwangsläufig auf Polymorphie, aber Polymorphie kann gewinnbringend eingesetzt werden, beispielsweise indem die View polymorph gehalten wird, so dass dasselbe Modell und derselbe Controller mit unterschiedlichen Views arbeiten können, einfach indem der polymorph gehaltene Typ des Views ausgetauscht wird.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Wozu Polymorphie: Separation of Concerns

- **Möglichst viele Typen gleichbehandeln an Stellen wo die Unterschiede zwischen den Typen nicht wichtig sind.**
- **Jede Einheit (Klasse, Subroutine...), deren Logik auf verschiedene Typen passt, möglichst nur einmal implementieren.**
  - **Die Unterschiede dann durch Füllen der offen gebliebenen Lücke.**
- **Beispiele:**

  foldr: (X Y -> Y) Y (list of X) -> Y

  Collections.sort (List<T>, Comparator<T>)

Es gibt in einem Stück Software häufig etliche Einheiten, die nicht nur auf einen einzelnen Typ oder eine einzelne Kombination von Typen passen, sondern auf viele gleichermaßen.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Wozu Polymorphie: Separation of Concerns

- **Möglichst viele Typen gleichbehandeln an Stellen wo die Unterschiede zwischen den Typen nicht wichtig sind.**
- **Jede Einheit (Klasse, Subroutine...), deren Logik auf verschiedene Typen passt, möglichst nur einmal implementieren.**
  - ➢ **Die Unterschiede dann durch Füllen der offen gebliebenen Lücke.**
- **Beispiele:**

  foldr: (X Y -> Y) Y (list of X) -> Y

  Collections.sort (List<T>, Comparator<T>)

Aus verschiedenen Gründen ist es natürlich hochgradig wünschenswert, eine solche Einheit nur einmal zu implementieren und sie durch Polymorphie für alle passenden Typen verfügbar zu machen.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Wozu Polymorphie: Separation of Concerns

- **Möglichst viele Typen gleichbehandeln an Stellen wo die Unterschiede zwischen den Typen nicht wichtig sind.**
- **Jede Einheit (Klasse, Subroutine...), deren Logik auf verschiedene Typen passt, möglichst nur einmal implementieren.**
  - ➢ **Die Unterschiede dann durch Füllen der offen gebliebenen Lücke.**
- **Beispiele:**

  foldr: (X Y -> Y) Y (list of X) -> Y

  Collections.sort (List<T>, Comparator<T>)

Das, was die einzelnen Typen voneinander unterscheidet, findet sich dann in den ausgelagerten Details wieder.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Wozu Polymorphie: Separation of Concerns

- **Möglichst viele Typen gleichbehandeln an Stellen wo die Unterschiede zwischen den Typen nicht wichtig sind.**
- **Jede Einheit (Klasse, Subroutine...), deren Logik auf verschiedene Typen passt, möglichst nur einmal implementieren.**
  - **Die Unterschiede dann durch Füllen der offen gebliebenen Lücke.**
- **Beispiele:**

  foldr: (X Y -> Y) Y (list of X) -> Y

  Collections.sort (List<T>, Comparator<T>)

Dieses Beispiel aus Racket hatten wir jetzt schon mehrfach diskutiert: Die beiden Typen X und Y sind offen gehalten, und die Details der Typen sind in den ersten Parameter ausgelagert.

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Wozu Polymorphie: Separation of Concerns

- **Möglichst viele Typen gleichbehandeln an Stellen wo die Unterschiede zwischen den Typen nicht wichtig sind.**
- **Jede Einheit (Klasse, Subroutine...), deren Logik auf verschiedene Typen passt, möglichst nur einmal implementieren.**
  - **Die Unterschiede dann durch Füllen der offen gebliebenen Lücke.**
- **Beispiele:**

  foldr: (X Y -> Y) Y (list of X) -> Y

  Collections.sort (List<T>, Comparator<T>)

Aber auch dieses Beispiel aus Java ist sicherlich einsichtsreich.

***Erinnerung*: Kapitel 06, Abschnitt zu Comparator.**

Abstraktion und Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Wozu Polymorphie: Separation of Concerns

- **Möglichst viele Typen gleichbehandeln an Stellen wo die Unterschiede zwischen den Typen nicht wichtig sind.**
- **Jede Einheit (Klasse, Subroutine...), deren Logik auf verschiedene Typen passt, möglichst nur einmal implementieren.**
  - **Die Unterschiede dann durch Füllen der offen gebliebenen Lücke.**
- **Beispiele:**

  foldr: (X Y -> Y) Y (list of X) -> Y

  Collections.sort (List<T>, Comparator<T>)

Neben dem generischen Typparameter T kommt durch Subtyppolymorphie noch eine frei wählbare Klasse hinzu unter allen, die das Interface Comparator mit Typ T implementieren. Der Aspekt oder Concern, wie zwei Elemente des Typs T miteinander verglichen werden sollen, wird aus der Sortiermethode ausgelagert, das ist hier die Anwendung des Prinzips Separation of Concerns. Dadurch muss man den Sortieralgorithmus nur einmal implementieren, und diese Implementation passt dann auf jeden beliebigen Typ T und jede beliebige Vergleichslogik auf T – geradezu ein Musterbeispiel für Abstraktion auf Typebene.

TECHNISCHE UNIVERSITÄT DARMSTADT

Ad-hoc Polymorphie: speziell Methodenüberladung

Ad-hoc Polymorphie ist eigentlich nur ein Randthema. Wir sagen nur ein paar Worte dazu und gehen kurz auf einen wichtigen Aspekt ein, der in Racket und Java *nicht* realisiert ist: Überladung von Operatoren.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
    public void m ( int n ) { ..... }
}

public class Y extends X {
    public void m ( double d ) { ..... }
    public void m ( String str ) { ..... }
}
```

In unserem ersten Beispiel wird wieder einmal eine Klasse Y von einer Klasse X abgeleitet.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public void m ( int n ) { ..... }
}

public class Y extends X {
   public void m ( double d ) { ..... }
   public void m ( String str ) { ..... }
}
```

Die Methode m kommt dreimal in Klasse Y vor, natürlich jeweils mit einer anderen Parameterliste: Eine Variante von Methode m ist von X geerbt, die anderen beiden sind erst in Y definiert. Wir sehen also nochmals hier, dass Vererbung auch zu Methodenüberladung führen kann.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
    public void m ( int n, double d ) { ..... }
}

public class Y extends X {
    public void m ( double d, int n ) { ..... }
    public void m ( String str, double d ) { ..... }
}
```

Das zweite Beispiel unterscheidet sich vom ersten Beispiel dadurch, dass jede Methode nun zwei Parameter hat.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
    public void m ( int n, double d ) { ..... }
}

public class Y extends X {
    public void m ( double d, int n ) { ..... }
    public void m ( String str, double d ) { ..... }
}
```

Wie wir wissen, ist es kein Problem, wenn derselbe formale Typ in verschiedenen Varianten derselben Methode auftritt, vorausgesetzt, die Parameterlisten sind *insgesamt* unterschiedlich. Hier sind zwar die Typen der Parameter dieselben, aber die Reihenfolge ist unterschiedlich.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
    public void m ( int n, double d ) { ..... }
}

public class Y extends X {
    public void m ( double d, int n ) { ..... }
    public void m ( String str, double d ) { ..... }
}
```

Natürlich auch bei diesem formalen Parametertyp. Der kommt zwar in Y zweimal an derselben Position vor, aber der andere Parameter ist jeweils unterschiedlich.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
class MyVector {
  private:
    double* the_vector_;
    .........
  public:
    inline int length() const { ......... }
    .........
}
```

Einige Sprachen wie beispielsweise C++ gehen noch einen Schritt weiter und erlauben neben der Überladung von Methoden auch die Überladung von Operatoren. Die Logik bei der Überladung von Operatoren ist völlig dieselbe wie bei der Überladung von Methoden, nur die Schreibweise ist anders. Daher reicht sicherlich ein einfaches Beispiel zur Illustration. Das Beispiel folgt dem Google C++ Style Guide.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
class MyVector {
  private:
    double* the_vector_;
    .........
  public:
    inline int length() const { ......... }
    .........
}
```

Klassen werden in C++ ähnlich definiert wie in Java und bedeuten auch ziemlich genau dasselbe. Für unsere Zwecke können die Unterschiede zwischen Klassen in C++ und Klassen in Java ignoriert werden. Der Google C++ Style Guide gibt vor, Klassennamen so zu bauen wie in Java, also alles zusammengeschrieben und jeder Wortanfang großgeschrieben.

***Nebenbemerkungen*: (1) Andere Style Guides weichen davon ab, und auch die C++-Standardbibliothek folgt einem anderen Style Guide. (2) Wird eine Klasse in C++ so wie hier nicht von einer anderen abgeleitet, dann ist sie tatsächlich von keiner anderen Klasse abgeleitet – im Gegensatz zu Java, wo sie implizit von java.lang.Object abgeleitet wäre.**

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
class MyVector {
  private:
    double* the_vector_;
    .........
  public:
    inline int length() const { ......... }
    .........
}
```

Ein kleiner notationeller Unterschied zu Java: In Java wird bei jedem einzelnen Attribut und bei jeder einzelnen Methode spezifiziert, ob dieses Attribut beziehungsweise diese Methode public, private oder protected ist. In C++ wird die Klassendefinition in entsprechende Bereiche unterteilt, und alle Attribute und Methoden in einem solchen Bereich haben dann das Zugriffsrecht gemäß diesem Bereich. Ein Bereich beginnt mit dem Schlüsselwort private, public oder protected, gefolgt von einem Doppelpunkt.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
class MyVector {
  private:
    double* the_vector_;
    .........
  public:
    inline int length() const { ......... }
    .........
}
```

Dieses Attribut ist also private.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
class MyVector {
  private:
    double* the_vector_;
    .........
  public:
    inline int length() const { ......... }
    .........
}
```

Der Google C++ Style Guide legt fest, dass Attribute im private-Bereich dieser Namenskonvention folgen: alle Wortbestandteile klein, Underscore zwischen zwei aufeinanderfolgenden Wortbestandteilen sowie ein Underscore am Ende, um private anzuzeigen.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
class MyVector {
  private:
    double* the_vector_;
    .........
  public:
    inline int length() const { ......... }
    .........
}
```

***Erinnerung*: In Kapitel 03b haben wir gesehen, dass Variablen von Arraytypen in Java nur die Adresse des eigentlichen Arrayobjektes enthalten. Während das in Java alles intern passiert, kann und muss man in C++ explizit mit Adressen umgehen. Der Fachbegriff dafür lautet Zeiger, englisch pointer. Dieses Attribut ist also ein Zeiger auf ein Array mit Komponententyp double und ist dafür da, später, nach seiner Initialisierung, die Anfangsaddresse eines double-Arrays zu speichern.**

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
class MyVector {
  private:
    double* the_vector_;
    .........
  public:
    inline int length() const { ......... }
    .........
}
```

Hier wird eine Methode der Klasse MyVector definiert.

***Nebenbemerkung*: Ein Array hat in C++ kein Attribut length oder ähnliches wie in Java. Die Länge des Arrays muss daher von Klasse MyVector selbst verwaltet werden. Die Implementation lassen wir hier aus.**

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
class MyVector {
  private:
    double* the_vector_;
    .........
  public:
    inline int length() const { ......... }
    .........
}
```

Das gibt es in Java nicht: Mit Schlüsselwort inline wird dem Compiler gesagt, dass er bei jedem Aufruf dieser Methode, den er im Quelltext findet, prüfen soll, ob er bei der Übersetzung hier wirklich den Code für einen Aufruf der Methode einfügen sollte oder nicht besser Code einfügt, der exakt dasselbe wie der Methodenaufruf leistet, aber ohne Methodenaufruf auskommt. Der Sinn ist natürlich Einsparung von Rechenzeit.

***Erinnerung*: Inlining hatten wir kurz in Kapitel 13, Abschnitt zu Laufzeitverbesserungen behandelt.**

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
class MyVector {
  private:
    double* the_vector_;
    .........
  public:
    inline int length() const { ......... }
    .........
}
```

Auch das gibt es in Java leider nicht: Mit Schlüsselwort const zwischen Methodenkopf und Methodenrumpf wird angezeigt, dass diese Methode das Objekt, mit dem sie aufgerufen wird, nicht verändert, also seine Attribute nicht überschreibt, Der Compiler sorgt dafür, dass die Methode dieses Versprechen auch einhält, das heißt, greift die Methode ändernd auf ihr Objekt zu, dann bricht der Compiler die Übersetzung mit einer Fehlermeldung ab.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
bool operator<= ( MyVector const& v ) const {
    bool return_value = true;
    for ( int i = 0; i <= length(); i++ )
        if ( the_vector[i] > v.the_vector[i] )
            return_value = false;
    return return_value;
}
```

Nach diesen Vorarbeiten kommen wir nun zum eigentlichen Thema dieses Beispiels: Überladung eines Operators für eine selbstdefinierte Klasse. Das ist eigentlich eine Methode der Klasse wie jede andere auch, nur mit einer einzigen Besonderheit.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
bool operator<= ( MyVector const& v ) const {
    bool return_value = true;
    for ( int i = 0; i <= length(); i++ )
        if ( the_vector[i] > v.the_vector[i] )
            return_value = false;
    return return_value;
}
```

Der Unterschied besteht im Namen der Methode. Der ist nicht mehr wie sonst im Rahmen der Regeln frei wählbar, sondern besteht zwingend aus dem Schlüsselwort operator, gefolgt von dem zu definierenden Operator.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
bool operator<= ( MyVector const& v ) const {
    bool return_value = true;
    for ( int i = 0; i <= length(); i++ )
        if ( the_vector[i] > v.the_vector[i] )
            return_value = false;
    return return_value;
}
```

Das Schlüsselwort const an dieser Stelle besagt, dass der aktuale Parameter nicht in der Methode geändert wird. Auch hier bricht der Compiler die Übersetzung mit einer Fehlermeldung ab, wenn die Methode das *doch* versucht. Auch das gibt es in Java nicht.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
bool operator<= ( MyVector const& v ) const {
    bool return_value = true;
    for ( int i = 0; i <= length(); i++ )
        if ( the_vector[i] > v.the_vector[i] )
            return_value = false;
    return return_value;
}
```

Und schließlich gibt es auch dies nicht in Java: In Kapitel 03b haben wir gesehen, dass immer nur die Adresse des Objektes kopiert wird. In C++ kann man sich das aussuchen: Mit dem Kaufmanns-Und sagt man, dass – wie in Java – nur die Adresse kopiert wird, ohne das Kaufmanns-Und wird das Objekt selbst kopiert.

***Nebenbemerkung*: Die Version mit Kaufmanns-Und nennt man in der Informatik call-by-reference, die ohne Kaufmanns-Und heißt call-by-value. Bei call-by-reference arbeitet die Methode also auf dem übergebenen Objekt; bei call-by-value arbeitet sie auf einer lokalen Kopie. In Java ist call-by-reference für Referenztypen und call-by-value für primitive Datentypen festgelegt, in anderen Sprachen wie etwa C++ kann der Entwickler dies für jeden Parameter individuell festlegen.**

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
bool operator<= ( MyVector const& v ) const {
    bool return_value = true;
    for ( int i = 0; i <= length(); i++ )
        if ( the_vector[i] > v.the_vector[i] )
            return_value = false;
    return return_value;
}
```

Der Datentyp mit den beiden Literalen true und false hat in C++ einen etwas kürzeren Namen als in Java.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
bool operator<= ( MyVector const& v ) const {
    bool return_value = true;
    for ( int i = 0; i <= length(); i++ )
        if ( the_vector[i] > v.the_vector[i] )
            return_value = false;
    return return_value;
}
```

Namen von lokalen Variablen werden nach dem Google C++ Style Guide dadurch gebildet, dass alle Buchstaben klein sind und zwischen zwei Wortbestandteilen jeweils ein Underscore steht, aber kein Underscore am Anfang oder Ende des Namens.

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
bool operator<= ( MyVector const& v ) const {
    bool return_value = true;
    for ( int i = 0; i <= length(); i++ )
        if ( the_vector[i] > v.the_vector[i] )
            return_value = false;
    return return_value;
}
```

Hier wird jetzt der boolesche Rückgabewert schrittweise berechnet. In C++ sieht dieses Fragment genau so aus, wie es in Java aussehen würde, so dass wohl nichts zum Code selbst gesagt werden muss. Die Logik des hier definierten Vergleichsoperators auf Arrays ist: Ein Array ist kleiner-gleich dem anderen, wenn jede Komponente des ersten Arrays kleiner-gleich der entsprechenden Komponente des zweiten Arrays ist.

***Nebenbemerkung*: Diese Vergleichslogik ist in verschiedensten Kontexten relevant, nämlich immer dann, wenn es um Optimierung mehrerer Kriterien gleichzeitig geht. Jedes Kriterium ist dann ein Arrayindex. Das Stichwort dazu in der Literatur ist *Pareto-Optimum*.**

Ad-hoc Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Operatorüberladung in C++:

```
MyVector v1 = .....;
MyVector v2 = .....;

.....

if ( v1 <= v2 ) .....
```

So wie auf dieser Folie kann der neu definierte Operator dann angewendet werden. Die Schreibweise ist identisch zu der bei eingebauten Datentypen.

***Nebenbemerkung*: In C++ kann man nur die Operatoren überladen, die schon für die primitiven Datentypen eingebaut sind, und auch nur mit derselben Parameterzahl, derselben Bindungsstärke und derselben Auswertungsreihenfolge von links nach rechts beziehungsweise von rechts nach links. In manchen anderen Sprachen gibt es mehr Freiraum bei der Überladung von Operatoren.**

***Erinnerung*: In Kapitel 01b gibt es einen Abschnitt zur Bindungsstärke von Operatoren in Java, der auch auf die Auswertungsreihenfolge eingeht. Dieser Abschnitt gilt im Wesentlichen auch für C und C++.**

TECHNISCHE UNIVERSITÄT DARMSTADT

Konzepte von Konformität

In den verschiedenen Konzepten von Polymorphie, die wir bisher betrachtet hatten, wurde Konformität jeweils unterschiedlich definiert und dementsprechend auch unterschiedlich abgeprüft, auch zu verschiedenen Zeitpunkten: Übersetzungszeit und Laufzeit.

Es ist wichtig, sich vor Augen zu halten, dass dies nur *mögliche* Varianten sind, wie man Konformität definieren kann. Um den Blick zu erweitern, müssen wir aber kurz in andere Programmiersprachen schauen.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Racket: ob die Operationen für die Operanden definiert sind**
- **Subtyppolymorphie: nur Funktionalität des *statischen* Typs erlaubt**
- **Ad-hoc Polymorphie:**
  - **Methodenüberladung: Signatur der Methode**
  - **Implizite Konversion: eingebaute Regeln**
- **Generizität: unterschiedliche Modelle in verschiedenen Sprachen**
  - **Wir vergleichen im Folgenden Java mit Ada und C++**

Wir beginnen mit einer Zusammenfassung, was wir bisher zu Konformität gesehen haben.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Racket: ob die Operationen für die Operanden definiert sind**
- **Subtyppolymorphie: nur Funktionalität des *statischen* Typs erlaubt**
- **Ad-hoc Polymorphie:**
  - **Methodenüberladung: Signatur der Methode**
  - **Implizite Konversion: eingebaute Regeln**
- **Generizität: unterschiedliche Modelle in verschiedenen Sprachen**
  - **Wir vergleichen im Folgenden Java mit Ada und C++**

Bei Racket haben wir gesehen, dass Konformität zur Laufzeit geprüft wird. Konkret zu prüfen sind Operationen, die nur für bestimmte Typen definiert sind. Das sind alles eingebaute Operationen in Form von Operatoren beziehungsweise vordefinierten Funktionen.

Natürlich kann man in Racket Funktionen schreiben, die nur für bestimmte Typen funktionieren, zum Beispiel nur für Zahlentypen. Aber wenn man genau hinschaut, stellt man fest, dass das Nichtfunktionieren einer selbstgeschriebenen Funktion für einen Typ immer darauf beruht, dass ein eingebauter Operator oder eine vordefinierte Funktion in der selbstgeschriebenen Funktion für diesen Typ verwendet wird, aber für diesen Typ nicht definiert ist.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

- Racket: ob die Operationen für die Operanden definiert sind
- Subtyppolymorphie: nur Funktionalität des ***statischen*** Typs erlaubt
- Ad-hoc Polymorphie:
  - Methodenüberladung: Signatur der Methode
  - Implizite Konversion: eingebaute Regeln
- Generizität: unterschiedliche Modelle in verschiedenen Sprachen
  - Wir vergleichen im Folgenden Java mit Ada und C++

Grundlegend für Subtyppolymorphie nicht nur in Java ist diese Regel aus Kapitel 03b: Der statische Typ definiert Konformität. Da alle Attribute und Methoden an die Subtypen vererbt werden, sind sie auf jeden Fall im dynamischen Typ vorhanden. Konformität ist damit garantiert.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

- Racket: ob die Operationen für die Operanden definiert sind
- Subtyppolymorphie: nur Funktionalität des ***statischen*** Typs erlaubt
- Ad-hoc Polymorphie:
  - Methodenüberladung: Signatur der Methode
  - Implizite Konversion: eingebaute Regeln
- Generizität: unterschiedliche Modelle in verschiedenen Sprachen
  - Wir vergleichen im Folgenden Java mit Ada und C++

Bei Methodenüberladung muss einfach nur ein Abgleich der Signaturen passieren, um zu prüfen, ob es genau eine passende Variante der Methode gibt.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

- Racket: ob die Operationen für die Operanden definiert sind
- Subtyppolymorphie: nur Funktionalität des ***statischen*** Typs erlaubt
- Ad-hoc Polymorphie:
  - Methodenüberladung: Signatur der Methode
  - Implizite Konversion: eingebaute Regeln
- Generizität: unterschiedliche Modelle in verschiedenen Sprachen
  - Wir vergleichen im Folgenden Java mit Ada und C++

Bei impliziter Konversion muss der Compiler nur schauen, ob eine solche zwischen den beiden beteiligten Datentypen vordefiniert ist.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Racket: ob die Operationen für die Operanden definiert sind**
- **Subtyppolymorphie: nur Funktionalität des *statischen* Typs erlaubt**
- **Ad-hoc Polymorphie:**
  - **Methodenüberladung: Signatur der Methode**
  - **Implizite Konversion: eingebaute Regeln**
- **Generizität: unterschiedliche Modelle in verschiedenen Sprachen**
  - **Wir vergleichen im Folgenden Java mit Ada und C++**

Generizität ist eine Fundgrube für verschiedene Konzepte von Konformität. Daher gehen wir darauf im Folgenden intensiver ein.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Racket: ob die Operationen für die Operanden definiert sind**
- **Subtyppolymorphie: nur Funktionalität des *statischen* Typs erlaubt**
- **Ad-hoc Polymorphie:**
  - **Methodenüberladung: Signatur der Methode**
  - **Implizite Konversion: eingebaute Regeln**
- **Generizität: unterschiedliche Modelle in verschiedenen Sprachen**
  - **Wir vergleichen im Folgenden Java mit Ada und C++**

Da es in wohl jeder Sprache höchstens eine Form von Generizität gibt, müssen wir uns mehrere Sprachen anschauen, um verschiedene Konzepte miteinander vergleichen zu können.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Generics in Java:

- **Ein Basistyp ist für eine generische Klasse / Methode auszuwählen.**
  - **Object falls nicht explizit ausgewählt.**
- **Nur die Funktionalität dieses Basistyps darf in der generischen Klasse / Methode verwendet werden.**
- **Nur der Basistyp und seine Subtypen dürfen die generische Klasse / Methode instanziieren.**

Als erstes schauen wir zurück, wie Konformität bei Generics in Java statisch geprüft wird.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Generics in Java:

- **Ein Basistyp ist für eine generische Klasse / Methode auszuwählen.**
  - ➢ **Object falls nicht explizit ausgewählt.**
- **Nur die Funktionalität dieses Basistyps darf in der generischen Klasse / Methode verwendet werden.**
- **Nur der Basistyp und seine Subtypen dürfen die generische Klasse / Methode instanziieren.**

***Erinnerung*: In Kapitel 06, Abschnitt zu eingeschränkten Typparametern, hatten wir schon das Konformitätskonzept von Generics in Java gesehen, nämlich indem der generische Typparameter auf einen Basistyp und seine Subtypen eingeschränkt wird.**

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Generics in Java:

- **Ein Basistyp ist für eine generische Klasse / Methode auszuwählen.**
  - ➢ **Object falls nicht explizit ausgewählt.**
- **Nur die Funktionalität dieses Basistyps darf in der generischen Klasse / Methode verwendet werden.**
- **Nur der Basistyp und seine Subtypen dürfen die generische Klasse / Methode instanziieren.**

Auch wenn man keinen einschränkenden Basistyp explizit hinschreibt, gibt es pro forma natürlich die Einschränkung auf die Klasse aller Klassen.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Generics in Java:

- **Ein Basistyp ist für eine generische Klasse / Methode auszuwählen.**
  - ➢ **Object falls nicht explizit ausgewählt.**
- **Nur die Funktionalität dieses Basistyps darf in der generischen Klasse / Methode verwendet werden.**
- **Nur der Basistyp und seine Subtypen dürfen die generische Klasse / Methode instanziieren.**

Das Konzept für Konformität bei Generics in Java ist übernommen von der Subtyppolymorphie, man kann mit einigem Recht daher von *einem* Konzept von Konformität für *beide* Formen von Polymorphie in Java sprechen.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Generics in Java:

- **Ein Basistyp ist für eine generische Klasse / Methode auszuwählen.**
  - ➢ **Object falls nicht explizit ausgewählt.**
- **Nur die Funktionalität dieses Basistyps darf in der generischen Klasse / Methode verwendet werden.**
- **Nur der Basistyp und seine Subtypen dürfen die generische Klasse / Methode instanziieren.**

Da nur der Basistyp und seine Subtypen als Typparameter in Frage kommen, ist Konformität garantiert.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Konformität ohne Basistyp: In einer generischen Einheit zu verwendende Funktionalität des polymorphen Typs ...

- **muss explizit in der generischen Einheit deklariert werden.**
  - **Z.B. in Ada → nächste Folie**
- **muss *nicht* deklariert werden.**
  - **Z.B. in C++ → übernächste Folie**

Dass Konformität auch grundsätzlich anders definiert sein kann und dass dann auch verschiedene Varianten möglich sind, werden wir jetzt anhand zweier anderer Programmiersprachen sehen.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Konformität ohne Basistyp: In einer generischen Einheit zu verwendende Funktionalität des polymorphen Typs ...

- **muss explizit in der generischen Einheit deklariert werden.**
  - **Z.B. in Ada → nächste Folie**
- **muss *nicht* deklariert werden.**
  - **Z.B. in C++ → übernächste Folie**

Durch Angabe eines Basistyps wird in Java explizit die Funktionalität, die der polymorphe Typ bereitzustellen hat, definiert. Mit Ada sehen wir gleich ein Beispiel dafür, dass eine solche explizite Definition auch anders möglich ist.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

Konformität ohne Basistyp: In einer generischen Einheit zu verwendende Funktionalität des polymorphen Typs ...

- **muss explizit in der generischen Einheit deklariert werden.**
  - **Z.B. in Ada → nächste Folie**
- **muss *nicht* deklariert werden.**
  - **Z.B. in C++ → übernächste Folie**

Und dass eine solche explizite Definition im Grunde gar nicht notwendig ist, werden wir danach bei C++ sehen.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In Ada:

```
generic
    type T is limited private;
    with function "*" ( X, Y: T ) return T;
    with function "+" ( X, Y: T ) return T;
    with function sqrt ( X: T ) return T;
function Euclidean_Distance ( X1, Y1, X2, Y2: T ) return T is
begin
    return sqrt ( X1 * X2 + Y1 * Y2 );
end
```

Zuerst also zu Generizität in Ada. Wie Sie sehen, unterscheidet sich die Syntax von Ada erheblich von der in Racket oder Java. Aber Sie sehen auch, dass Ada offensichtlich auf leichte Verständlichkeit ausgelegt ist. Mit ein paar oberflächlichen Erläuterungen, die jetzt folgen, sollte dieses kleine Beispiel daher problemlos verstehbar sein.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In Ada:

```
generic
    type T is limited private;
    with function "*" ( X, Y: T ) return T;
    with function "+" ( X, Y: T ) return T;
    with function sqrt ( X: T ) return T;
function Euclidean_Distance ( X1, Y1, X2, Y2: T ) return T is
begin
    return sqrt ( X1 * X2 + Y1 * Y2 );
end
```

Wie in Racket, sind Subroutinen in Ada nicht unbedingt an Klassen oder ähnliches gebunden.

***Nebenbemerkung*: In Ada wird zwischen Funktionen und Prozeduren unterschieden: Eine Funktion hat einen Rückgabewert, eine Prozedur nicht. Prozeduren sind daher vergleichbar mit void-Methoden in Java.**

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In Ada:

```
generic
    type T is limited private;
    with function "*" ( X, Y: T ) return T;
    with function "+" ( X, Y: T ) return T;
    with function sqrt ( X: T ) return T;
function Euclidean_Distance ( X1, Y1, X2, Y2: T ) return T is
begin
    return sqrt ( X1 * X2 + Y1 * Y2 );
end
```

Einzelne Parameter einer Subroutine werden etwas anders als in Java deklariert: erst der Name des Parameters, dann sein Typ, getrennt durch einen Doppelpunkt.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In Ada:

```
generic
    type T is limited private;
    with function "*" ( X, Y: T ) return T;
    with function "+" ( X, Y: T ) return T;
    with function sqrt ( X: T ) return T;
function Euclidean_Distance ( X1, Y1, X2, Y2: T ) return T is
begin
    return sqrt ( X1 * X2 + Y1 * Y2 );
end
```

Wie in Java, werden die einzelnen Parameter durch Komma voneinander getrennt. Im Unterschied zu Java muss bei mehreren Parametern, die denselben Typ haben und in der Parameterliste aufeinanderfolgen, der Typ nur einmal ganz am Ende aufgeführt werden. Hier werden also vier Parameter des Typs T deklariert. Das sind natürlich die Koordinaten der beiden Punkte, von denen die euklidische Distanz berechnet werden soll.

***Nebenbemerkung*: Diese Definition der Euklidischen Distanz ist nicht die übliche für komplexe Zahlen. Da man sicherlich auch komplexe Zahlentypen als Instanziierungen für T zulassen möchte, müsste man eigentlich die Beträge der beiden Produkte addieren, nicht wie hier die Produkte selbst. Aber hier kommt es ja nur auf das Prinzip an.**

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In Ada:

```
generic
    type T is limited private;
    with function "*" ( X, Y: T ) return T;
    with function "+" ( X, Y: T ) return T;
    with function sqrt ( X: T ) return T;
function Euclidean_Distance ( X1, Y1, X2, Y2: T ) return T is
begin
    return sqrt ( X1 * X2 + Y1 * Y2 );
end
```

Auch die Angabe des Rückgabetyps ist in Ada leicht anders als in Java: erst am Ende des Funktionskopfes und mit Schlüsselwort return.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In Ada:

```
generic
    type T is limited private;
    with function "*" ( X, Y: T ) return T;
    with function "+" ( X, Y: T ) return T;
    with function sqrt ( X: T ) return T;
function Euclidean_Distance ( X1, Y1, X2, Y2: T ) return T is
begin
    return sqrt ( X1 * X2 + Y1 * Y2 );
end
```

Anstelle von geschweiften Klammern müssen Sie in Ada diese drei Schlüsselwörter rund um den Methodenrumpf schreiben.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In Ada:

```
generic
    type T is limited private;
    with function "*" ( X, Y: T ) return T;
    with function "+" ( X, Y: T ) return T;
    with function sqrt ( X: T ) return T;
function Euclidean_Distance ( X1, Y1, X2, Y2: T ) return T is
begin
    return sqrt ( X1 * X2 + Y1 * Y2 );
end
```

Natürlich gibt es auch bei den Anweisungen in Ada Unterschiede zu Java, teilweise auch große. Aber in diesem kleinen Beispiel ist alles wie in Java.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In Ada:

```
generic
    type T is limited private;
    with function "*" ( X, Y: T ) return T;
    with function "+" ( X, Y: T ) return T;
    with function sqrt ( X: T ) return T;
function Euclidean_Distance ( X1, Y1, X2, Y2: T ) return T is
begin
    return sqrt ( X1 * X2 + Y1 * Y2 );
end
```

Hier wird der generische Typparameter deklariert.

***Nebenbemerkung*: Die Deklaration als limited und private ist wichtig, damit diese Funktion für möglichst viele Typen verfügbar ist. Eine genauere Erläuterung ist für das Verständnis unseres Themas nicht notwendig und würde daher zu weit führen.**

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In Ada:

```
generic
    type T is limited private;
    with function "*" ( X, Y: T ) return T;
    with function "+" ( X, Y: T ) return T;
    with function sqrt ( X: T ) return T;
function Euclidean_Distance ( X1, Y1, X2, Y2: T ) return T is
begin
    return sqrt ( X1 * X2 + Y1 * Y2 );
end
```

Hier kommen wir nun zum eigentlichen Thema. In der hier definierten Funktion Euclidean_Distance wird eine Funktion namens sqrt verwendet. Diese muss nach der Deklaration des generischen Typparameters, aber vor der eigentlichen Definition der Subroutine explizit deklariert werden, und zwar mit Schlüsselwort with.

Es wird also nicht wie in Java gesagt, dass T ein bestimmter Typ sein soll, etwas ein Subtyp eines bestimmten Basistyps. Sondern es wird einfach nur gesagt, dass eine bestimmte Subroutine mit genau festgelegtem Namen, genau festgelegter Parameterliste und – bei Funktionen – auch genau festgelegtem Rückgabetyp für den generischen Typparameter vorhanden sein muss.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In Ada:

```
generic
    type T is limited private;
    with function "*" ( X, Y: T ) return T;
    with function "+" ( X, Y: T ) return T;
    with function sqrt ( X: T ) return T;
function Euclidean_Distance ( X1, Y1, X2, Y2: T ) return T is
begin
    return sqrt ( X1 * X2 + Y1 * Y2 );
end
```

Hier sehen Sie dann noch, dass auch Ada die Überladung von Operatoren erlaubt. Ist T ein eingebauter numerischer Typ, dann sind diese beiden Operatoren von vornherein vorhanden. Für andere Typen T muss man sie mit dieser Parameterliste und diesem Rückgabetyp überladen.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In C++:

```
template <class T> void demonstrate_the_concept ( T t ) {
   t.make_something_reasonable ( 123, ´a´ );
}

class my_class {
   public: void make_something_reasonable ( int n, char c );
}

my_class my_object ();
demonstrate_the_concept ( my_object );
```

Caveat: kein Beispiel für guten C++-Stil!

Wie angekündigt, wenden wir uns nun C++ zu, um ein Beispiel dafür zu sehen, dass die erwartete Funktionalität gar nicht deklariert werden muss – weder durch einen Basistyp wie in Java noch durch Deklaration der verwendeten Funktionalität in der generischen Einheit wie in Ada noch sonst irgendwie.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In C++:

```
template <class T> void demonstrate_the_concept ( T t ) {
   t.make_something_reasonable ( 123, ´a´ );
}

class my_class {
   public: void make_something_reasonable ( int n, char c );
}

my_class my_object ();
demonstrate_the_concept ( my_object );
```

Caveat: kein Beispiel für guten C++-Stil!

Neben Methoden von Klassen erlaubt C++ auch unabhängige Subroutinen wie in Racket und Ada. Die Syntax ist in diesem Beispiel identisch mit der in Java, und auch über dieses Beispiel hinaus sind die Unterschiede in der Syntax von Methodenköpfen zwischen Java und C++ eher gering.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In C++:

```
template <class T> void demonstrate_the_concept ( T t ) {
   t.make_something_reasonable ( 123, ´a´ );
}

class my_class {
   public: void make_something_reasonable ( int n, char c );
}

my_class my_object ();
demonstrate_the_concept ( my_object );
```

Caveat: kein Beispiel für guten C++-Stil!

Die Deklarierung der Subroutine als generisch sieht in C++ allerdings völlig anders aus. Das generische Konzept heißt bei C++ auch nicht *Generics*, sondern *Templates*.

***Nebenbemerkung*: In C++ müssen generische Parameter nicht Typparameter sein, sondern dürfen auch ganze Zahlen sein, so dass man eine generische Einheit mit ganzzahligem generischem Parameter mit jeder beliebigen ganzen Zahl instanziieren kann. Das Schlüsselwort class hier sagt an, dass T tatsächlich ein Typparameter ist, wie Sie es gewohnt sind.**

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In C++:

```
template <class T> void demonstrate_the_concept ( T t ) {
  t.make_something_reasonable ( 123, ´a´ );
}

class my_class {
   public: void make_something_reasonable ( int n, char c );
}

my_class my_object ();
demonstrate_the_concept ( my_object );
```

Caveat: kein Beispiel für guten C++-Stil!

In dieser Anweisung wird eine Methode von T verwendet. Jeder Typ, der T instanziieren soll, muss also eine Klasse sein, die diese Methode mit genau diesem Namen und genau zwei Parametern hat, wobei der erste Parameter entweder von Typ int oder ein Typ sein muss, in den int implizit konvertiert wird, denn hier wird ja ein int übergeben. Analog muss der zweite Parameter vom Typ char oder einem Typ sein, in den char implizit konvertiert wird.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In C++:

```
template <class T> void demonstrate_the_concept ( T t ) {
   t.make_something_reasonable ( 123, ´a´ );
}
class my_class {
   public: void make_something_reasonable ( int n, char c );
}
my_class my_object ();
demonstrate_the_concept ( my_object );
```

Caveat: kein Beispiel für guten C++-Stil!

Hier definieren wir nun eine Klasse, mit der T oben in der generischen Subroutine beispielhaft instanziiert werden soll.

***Erinnerung*: Weiter vorne, im Abschnitt zu Ad-hoc Polymorphie hatten wir in einem Beispiel zur Operatorüberladung schon einiges zu C++ gesehen, was Sie jetzt hier wiedersehen und daher nicht noch einmal erläutert wird.**

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In C++:

```
template <class T> void demonstrate_the_concept ( T t ) {
   t.make_something_reasonable ( 123, ´a´ );
}

class my_class {
  public: void make_something_reasonable ( int n, char c );
}

my_class my_object ();
demonstrate_the_concept ( my_object );
```

Caveat: kein Beispiel für guten C++-Stil!

Hier wird die Methode eingeführt, die in der generischen Subroutine dann aufgerufen wird.

***Nebenbemerkung*: Auch wenn Sie hier keine Implementation der Methode sehen, ist diese Methode dennoch nicht abstract. In C++ muss eine Methode nicht direkt in der Definition der Klasse implementiert werden, sondern die Implementation kann auch außerhalb der Klassendefinition – auch in einer anderen Quelldatei – stehen. Das ist sogar der Normalfall.**

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In C++:

```
template <class T> void demonstrate_the_concept ( T t ) {
   t.make_something_reasonable ( 123, ´a´ );
}

class my_class {
   public: void make_something_reasonable ( int n, char c );
}

my_class my_object ();
demonstrate_the_concept ( my_object );
```

Caveat: kein Beispiel für guten C++-Stil!

Hier wird ein Objekt dieser Klasse eingerichtet.

***Nebenbemerkung*: In C++ haben Sie als Entwickler die Wahl, ob Sie ein Objekt wie in Java mit Operator new einrichten oder so wie hier gezeigt. Wird die hier farblich unterlegte Form *ohne* new gewählt, dann wird das Objekt im Frame dieser Methode auf dem Call-Stack abgelegt und daher auch nach Beendigung der Methode automatisch wieder abgebaut. Wie bei primitiven Datentypen gibt es in diesem Fall keine Unterscheidung zwischen Referenz und Objekt.**

***Erinnerung*: Kapitel 01e, Abschnitt zur Ausführung von Methoden und zum Call-Stack.**

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In C++:

```
template <class T> void demonstrate_the_concept ( T t ) {
   t.make_something_reasonable ( 123, ´a´ );
}

class my_class {
   public: void make_something_reasonable ( int n, char c );
}

my_class my_object ();
demonstrate_the_concept ( my_object );
```

Caveat: kein Beispiel für guten C++-Stil!

Hier wird nun die generische Subroutine mit einem Objekt der beispielhaften Klasse aufgerufen.

Der entscheidende Punkt ist, dass der Compiler bei der Übersetzung an dieser Stelle einfach so prüft, ob die Klasse tatsächlich die Methode hat, die in der generischen Subroutine aufgerufen wird, und ob diese Methode konform zu ihrer Definition verwendet wird. Weder ist die Klasse wie in Java von einem bestimmten Typ abgeleitet worden, um Konformität zu garantieren, noch ist wie in Ada irgendwo die verwendete Funktionalität deklariert.

Gegenüber Ada spart das dem Entwickler natürlich einen gewissen, nicht allzu großen Schreibaufwand. *Ein* Nachteil des Konzepts in C++ ist offensichtlich: Der Entwickler der generischen Einheit muss so diszipliniert sein, dass er das, was in Ada in der with-Klausel steht, dann wenigstens in einen Kommentar schreibt, damit andere Entwickler wissen, wie sie eine Klasse konzipieren müssen, um die generische Einheit auf diese Klasse anzuwenden. Damit ist die ohnehin schon geringe Reduktion des Schreibaufwands natürlich perdu.

Konzepte von Konformität

TECHNISCHE UNIVERSITÄT DARMSTADT

In C++:

```
template <class T> void demonstrate_the_concept ( T t ) {
   t.make_something_reasonable ( 123, ´a´ );
}

class my_class {
   public: void make_something_reasonable ( int n, char c );
}

my_class my_object ();
demonstrate_the_concept ( my_object );
```

Caveat: kein Beispiel für guten C++-Stil!

Ein *zweiter* Nachteil wird sichtbar, wenn man intensiv mit Templates in C++ arbeitet, vor allem wenn man komplexere, ineinander verschachtelte Template-Konstruktionen entwickelt. Wenn man irgendwo Fehler einbaut, dann kann der Ada-Compiler den Fehler sehr genau lokalisieren, denn dank der with-Klauseln kann er entscheiden, ob der Fehler in der *Implementation* oder in der Anwendung der generischen Einheit passiert ist: Im ersten Fall passt eben die Implementation nicht zu den eigenen with-Klauseln, im zweiten Fall die Anwendung. In C++ hingegen weiß der Compiler dazu gar nichts. Das äußert sich in der Praxis in zuweilen unglaublich langen, hochgradig unübersichtlichen Fehlermeldungen.

TECHNISCHE UNIVERSITÄT DARMSTADT

Multiple Polymorphie

Bisher haben wir einzelne polymorphe Typen mehr oder weniger unabhängig voneinander betrachtet. Es gibt aber polymorphe Konzepte, bei denen die Bindung von mehr als einem beteiligten polymorphen Typ abhängt. Das schauen wir uns in diesem Abschnitt genauer an.

Multiple Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Generics in Java:

```
public class ABPred <T1, T2>
  implements BiPredicate  <X extends A<T1>, Y extends B<T2>> {
    boolean test ( X x, Y y ) {
      return x.f() == y.g();
    }
}
```

```
public class A <T> {
    T f () { ..... }
}
```

```
public class B <T> {
    T g () { ..... }
}
```

Multiple Polymorphie haben wir eigentlich schon gesehen, nämlich bei Generics in Java, wir hatten das Thema multiple Polymorphie bislang halt nicht explizit behandelt. Auf dieser Folie sehen Sie beispielhaft eine Klasse, die das generische Interface BiPredicate aus dem Package java.util.function implementiert.

Multiple Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Generics in Java:

```
public class ABPred <T1, T2>
  implements BiPredicate <X extends A<T1>, Y extends B<T2>> {
    boolean test ( X x, Y y ) {
      return x.f() == y.g();
    }
}
```

```
public class A <T> {
   T f () { ..... }
}
```

```
public class B <T> {
   T g () { ..... }
}
```

Dies ist ein einfaches, rein illustratives Beispiel dafür, dass die Typparameter konform zueinander sein müssen. Der Compiler prüft, dass X und Y gleich A beziehungsweise B oder Subtypen davon sind, so dass die beiden aufgerufenen Methoden tatsächlich existieren.

Dabei müssen nicht nur X und Y mit dem generischen Kontext, sondern auch zueinander konform sein, nämlich beim Test auf Gleichheit, denn da müssen die Typen auf beiden Seiten ja identisch sein oder zumindest in einem Subtyp-Supertyp-Verhältnis zueinander stehen. Konkret läuft das in diesem Beispiel darauf hinaus, dass T1 und T2 identisch sein müssen. Der Compiler prüft das und bricht die Übersetzung gegebenenfalls mit einer Fehlermeldung ab.

Multiple Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Multi-Methoden (multiple dispatch) in Common Lisp:

```
( defmethod translate ( ( x german-text ) ( y english-text ) ) ..... )
( defmethod translate ( ( x english-text ) ( y german-text ) ) ..... )
( defmethod translate ( ( x italian-text  ) ( y german-text ) ) ..... )
( defmethod translate ( ( x english-text ) ( y french-text  ) ) ..... )
```

```
( translate a b )
```

Jede Objektmethode hängt in Java an einem einzelnen Objekt, und der Typ dieses Objektes entscheidet, welche Implementation dieser Methode angesteuert wird. Mehr geht in Java nicht.

Es gibt aber auch Programmiersprachen, in denen bei Subtyppolymorphie mehr möglich ist, zum Beispiel Common Lisp. Das Stichwort in der Literatur lautet Multi-Methoden oder auch multiple dispatch. Wir sehen uns das nur ganz kurz auf dieser Folie an einem durchaus realistischen Beispiel an.

Multiple Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Multi-Methoden (multiple dispatch) in Common Lisp:

```
( defmethod translate ( ( x german-text ) ( y english-text ) ) ..... )
( defmethod translate ( ( x english-text ) ( y german-text ) ) ..... )
( defmethod translate ( ( x italian-text  ) ( y german-text ) ) ..... )
( defmethod translate ( ( x english-text ) ( y french-text  ) ) ..... )
```

```
( translate a b )
```

Wir nehmen an, dass wir eine Reihe von Klassen geschrieben haben für Texte in unterschiedlichen Sprachen.

Multiple Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Multi-Methoden (multiple dispatch) in Common Lisp:

```
( defmethod translate ( ( x german-text ) ( y english-text ) ) ..... )
( defmethod translate ( ( x english-text ) ( y german-text ) ) ..... )
( defmethod translate ( ( x italian-text  ) ( y german-text ) ) ..... )
( defmethod translate ( ( x english-text ) ( y french-text  ) ) ..... )
```

```
( translate a b )
```

Für diverse Kombinationen von Sprachen schreiben wir jeweils eine Methode zur Übersetzung von der einen in die andere Sprache. Alle diese Methoden nennen wir gleich, nämlich translate.

Multiple Polymorphie

TECHNISCHE UNIVERSITÄT DARMSTADT

Multi-Methoden (multiple dispatch) in Common Lisp:

```
( defmethod translate ( ( x german-text ) ( y english-text ) ) ..... )
( defmethod translate ( ( x english-text ) ( y german-text ) ) ..... )
( defmethod translate ( ( x italian-text  ) ( y german-text ) ) ..... )
( defmethod translate ( ( x english-text ) ( y french-text  ) ) ..... )
```

```
( translate a b )
```

Wenn wir nun translate aufrufen mit zwei Texten von unbekannten dynamischen Typen, dann steuert das Laufzeitsystem die passende Implementation abhängig von *beiden* beteiligten Typen an, also tatsächlich die Übersetzung aus der Sprache von a in die Sprache von b.

In Java und den meisten anderen gängigen objektorientierten Sprachen wäre dies nur mit *einem* der beiden beteiligten Typen möglich. Die Methode würde dann so, wie wir es bisher in Java gesehen haben, mit einem Objekt des *einen* beteiligten Typs aufgerufen, und das Objekt des *anderen* beteiligten Typs müsste dann ein aktualer Parameter der Methode sein. So lässt sich aber natürlich nicht realisieren, dass die Auswahl der Implementation der Methode wie in Common Lisp von *beiden* Typen abhängt.

TECHNISCHE UNIVERSITÄT DARMSTADT

Duck-Typing objektorientiert

Zum Abschluss dieses Abschnitts und auch des ganzen Kapitels kommen wir noch einmal kurz auf rein dynamische Polymorphie zurück, also Duck-Typing. Das haben wir bisher in Racket kennen gelernt. Manchmal wird Duck-Typing fälschlich als eine Eigenart von funktionalen oder allgemeiner deklarativen Sprachen angesehen.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Nachrichten in Smalltalk:

```
9 sqrt
'Hello World' size
'Hello World' at: 5
myNonnegativeNumber sqrt
myString size
myString at: 357
myArray at: 357 put: 'Hello World'
```

Um zu zeigen, dass dem nicht so ist, schauen wir uns zwei Beispiele aus objektorientierten Sprachen an, zuerst ein einfaches Konzept in Smalltalk, dann ein komplizierteres, bislang noch nicht gesehenes Konzept in Java.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Nachrichten in Smalltalk:

```
9 sqrt
'Hello World' size
'Hello World' at: 5
myNonnegativeNumber sqrt
myString size
myString at: 357
myArray at: 357 put: 'Hello World'
```

Fangen wir mit einem möglichst einfachen Beispiel an.

In Smalltalk gibt es nur Objekte, sonst nichts. Auch ein Zahlenliteral ist ein Objekt. Einem Objekt kann man Nachrichten senden. Hier wird dem Objekt, das den Namen 9 hat, eine Nachricht namens sqrt gesendet. Zur Laufzeit wird geprüft, ob der Typ des Objekts namens 9 tatsächlich eine Methode namens sqrt hat. In typischer Smalltalk-Sprechweise würde man dasselbe so ausdrücken: Es wird geprüft, ob dieses Objekt diese Nachricht tatsächlich auch empfangen kann. Das Ergebnis dieser Nachrichtensendung ist natürlich 3.

Wenn das der Fall ist, wird die Methode für das Objekt ausgeführt, ansonsten gibt es eine Fehlermeldung und Programmabbruch.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Nachrichten in Smalltalk:

```
9 sqrt
'Hello World' size
'Hello World' at: 5
myNonnegativeNumber sqrt
myString size
myString at: 357
myArray at: 357 put: 'Hello World'
```

Strings sind ebenfalls Objekte. Wie Sie sehen, werden Strings in Smalltalk in einfache Hochkommas gesetzt. Hier wird die Nachricht size diesem String-Objekt gesendet. Das Ergebnis dieser Nachrichtensendung ist 11.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Nachrichten in Smalltalk:

```
9 sqrt
'Hello World' size
'Hello World' at: 5
myNonnegativeNumber sqrt
myString size
myString at: 357
myArray at: 357 put: 'Hello World'
```

So sieht es aus, wenn die Nachricht einen Parameter hat: Name der Nachricht und dann der Parameterwert, getrennt durch einen Doppelpunkt.

Dieser Ausdruck liefert den Wert am Index 5 des Strings zurück. Da Indizes in Smalltalk mit 1 beginnen, ist das also das kleine o.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Nachrichten in Smalltalk:

```
9 sqrt
'Hello World' size
'Hello World' at: 5
myNonnegativeNumber sqrt
myString size
myString at: 357
myArray at: 357 put: 'Hello World'
```

Und natürlich funktioniert das alles exakt genauso mit *Variablen* von diesen Typen, nicht nur mit *Literalen*.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Nachrichten in Smalltalk:

```
9 sqrt
'Hello World' size
'Hello World' at: 5
myNonnegativeNumber sqrt
myString size
myString at: 357
myArray at: 357 put: 'Hello World'
```

Das geht jetzt über unser eigentliches Thema hinaus, soll aber nicht unerwähnt bleiben, weil es wunderbar unsere durch Racket und Java geschulte Vorstellung, wie die Dinge so sind, hinterfragt, hier speziell unsere Vorstellung von Subroutinen und ihren Parametern. Das Stichwort für weitere Recherche lautet keyword messages.

Die Nachricht, die hier an myArray gesendet wird, hat den Namen „at.put“. Man kann also eine Nachricht aus mehreren Bestandteilen zusammensetzen, die durch Punkte voneinander getrennt sind. Jeder Parameter dieser Nachricht ist an genau einen Bestandteil gebunden. Die Idee dahinter, die in der Praxis auch gut funktioniert, ist, dass eine solche Anweisung besser lesbar wird, einfach weil man gezwungen ist, den Namen der Nachricht so zu wählen, dass die Namensbestandteile geeignet mit den einzelnen Parametern verbunden werden können.

Im konkreten Beispiel wird die Komponente von myArray an Index 357 mit dem Parameter von put überschrieben – englisch formuliert: at (index) 357 put (the string) “Hello World“.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Reflection in Java: import java.lang.reflect.*;

```
public static void demonstrateReflection ( String nameOfSomeClass ) {
   Integer i = 123;
   String str = "Hello World";
   Class<?> c = Class.forName ( nameOfSomeClass );
   try {
      Method m = c.getDeclaredMethod ( "myMethod", i.getClass(), str.getClass() );
   }
   catch ( NoSuchMethodException exc ) { ..... }
   .....
}
```

Zu guter Letzt kommen wir wie angekündigt zu einer Form von Duck Typing in Java. Das Stichwort dazu lautet Reflection. Reflection bietet umfassende Möglichkeiten, ein Objekt von einem unbekannten Typ zu analysieren und zu manipulieren, unter anderem seine Methoden aufzurufen. Oben sehen Sie das zentrale Package dafür.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Reflection in Java: import java.lang.reflect.*;

```
public static void demonstrateReflection ( String nameOfSomeClass ) {
   Integer i = 123;
   String str = "Hello World";
   Class<?> c = Class.forName ( nameOfSomeClass );
   try {
      Method m = c.getDeclaredMethod ( "myMethod", i.getClass(), str.getClass() );
   }
   catch ( NoSuchMethodException exc ) { ..... }
   .....
}
```

In Package java.lang findet sich eine Klasse mit Namen Class, die vielfältige Funktionalität rund um Klassen anbietet. Zu jeder Klasse X gibt es im Laufzeitsystem ein Objekt vom Typ Class, wobei der Typparameter von Class mit X instanziiert ist, also Class<X>. Die hier aufgerufene Klassenmethode forName liefert das Class-Objekt für die im Parameter spezifizierte Klasse zurück. Der Parameter muss den voll qualifizierten Namen der Klasse angeben, also inklusive Package-Pfad.

Darüber hinaus gibt es für jeden primitiven Datentyp jeweils ein Class-Objekt sowie auch für das Schlüsselwort void. Der Sinn wird gleich klar werden.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Reflection in Java: import java.lang.reflect.*;

```
public static void demonstrateReflection ( String nameOfSomeClass ) {
   Integer i = 123;
   String str = "Hello World";
   Class<?> c = Class.forName ( nameOfSomeClass );
   try {
      Method m = c.getDeclaredMethod ( "myMethod", i.getClass(), str.getClass() );
   }
   catch ( NoSuchMethodException exc ) { ..... }
   .....
}
```

Von einem solchen Klassenobjekt kann man sich dann auch einzelne Methoden der zugehörigen Klasse in Form von Objekten einer Klasse namens Method aus Package java.lang.reflect zurückliefern lassen. Der erste Parameter ist der Name der Methode, für die ein Method-Objekt zurückgeliefert werden soll.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Reflection in Java: import java.lang.reflect.*;

```
public static void demonstrateReflection ( String nameOfSomeClass ) {
   Integer i = 123;
   String str = "Hello World";
   Class<?> c = Class.forName ( nameOfSomeClass );
   try {
      Method m = c.getDeclaredMethod ( "myMethod", i.getClass(), str.getClass() );
   }
   catch ( NoSuchMethodException exc ) { ..... }
   .....
}
```

Die Methode getDeclaredMethod kann beliebig viele weitere Parameter haben, auch keinen. In diesem kleinen Beispiel hat sie *zwei* weitere Parameter. Sie muss genauso viele weitere Parameter neben dem Methodennamen haben, wie die zurückzuliefernde Methode insgesamt an Parametern hat. Jeder Parameter ist das Class-Objekt des Parametertyps an dieser Stelle der Parameterliste. Dafür hat Klasse Object eine Methode getClass, die wie immer an alle anderen Klassen vererbt wird. Sie liefert das Class-Objekt zur Klasse des Objektes zurück, mit dem getClass aufgerufen wird.

Hier sollte jetzt klarwerden, warum ein Class-Objekt auch für jeden primitiven Datentyp vorhanden sein muss, denn sonst könnte man ja gar nicht auf diese Art auf Methoden mit Parametern von primitiven Datentypen zugreifen. Dass es auch ein Class-Objekt speziell für void gibt, liegt daran, dass es auch Möglichkeiten gibt, mit Rückgabetypen von Methoden zu arbeiten, und der Rückgabetyp einer Methode kann ja formal auch void sein.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Reflection in Java: import java.lang.reflect.*;

```
public static void demonstrateReflection ( String nameOfSomeClass ) {
   Integer i = 123;
   String str = "Hello World";
   Class<?> c = Class.forName ( nameOfSomeClass );
   try {
      Method m = c.getDeclaredMethod ( "myMethod", i.getClass(), str.getClass() );
   }
   catch ( NoSuchMethodException exc ) { ..... }
   .....
}
```

Natürlich kann es sein, dass die Klasse gar keine so spezifizierte Methode hat. In diesem Fall wird eine Exception geworfen, deren Klasse kein Subtyp von RuntimeException ist und daher gefangen oder weitergereicht werden muss.

***Achtung*: Methode getDeclaredMethod wirft potentiell auch eine SecurityException. Die ist aber von RuntimeException abgeleitet und darf daher ignoriert werden – wie immer auf eigene Gefahr.**

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Reflection in Java: import java.lang.reflect.*;

```
public static void demonstrateReflection ( String nameOfSomeClass ) {
   Integer i = 123;
   String str = "Hello World";
   Class<?> c = Class.forName ( nameOfSomeClass );
   try {
      Method m = c.getDeclaredMethod ( "myMethod", i.getClass(), str.getClass() );
   }
   catch ( NoSuchMethodException exc ) { ..... }
   .....
}
```

Den Rest des Codes schauen wir uns auf der nächsten Folie an.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Reflection in Java: import java.lang.reflect.*;

```
public static void demonstrateReflection ( String nameOfSomeClass ) {
   Integer i = 123;
   String str = “Hello World“;
   Class<?> c = Class.forName ( nameOfSomeClass );
   .....
   try {
      m.invoke ( i, str );
   }
   catch ( IllegalAccessException exc ) { ..... }
   catch (  InvocationTargetException exc ) { ..... }
}
```

Den Aufruf der Methode getDeclaredMethod nebst try-catch-Block von der letzten Folie lassen wir aus Platzgründen aus.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Reflection in Java: import java.lang.reflect.*;

```
public static void demonstrateReflection ( String nameOfSomeClass ) {
   Integer i = 123;
   String str = "Hello World";
   Class<?> c = Class.forName ( nameOfSomeClass );
   .....
   try {
      m.invoke ( i, str );
   }
   catch ( IllegalAccessException exc ) { ..... }
   catch (  InvocationTargetException exc ) { ..... }
}
```

Mit dieser Anweisung rufen wir die Methode auf, für die wir mit getDeclaredMethod eben ein Objekt der Klasse Method zurückerhalten hatten. Anzahl und Typen der Parameter müssen natürlich stimmen.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Reflection in Java: import java.lang.reflect.*;

```
public static void demonstrateReflection ( String nameOfSomeClass ) {
   Integer i = 123;
   String str = "Hello World";
   Class<?> c = Class.forName ( nameOfSomeClass );
   .....
   try {
      m.invoke ( i, str );
   }
   catch ( IllegalAccessException exc ) { ..... }
   catch (  InvocationTargetException exc ) { ..... }
}
```

Auch die Methode invoke von Klasse Method wirft potentiell Exceptions, deren Klassen keine Subtypen von RuntimeException sind.

***Achtung*: Hinzu kommen eine weitere Exception-Klasse, die ein Subtyp von RuntimeException ist, und sogar eine Error-Klasse.**

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Reflection in Java: import java.lang.reflect.*;

```
public static int demonstrateDuckTyping ( Object obj ) {
   Integer i = 123;
   String str = “Hello World“;
   Class<?> c = obj.getClass();
    // On last slide: Class<?> c = Class.forName ( nameOfSomeClass );

   .....
}
```

Nun haben wir alles zusammen, um Duck Typing mittels Reflection in Java zu demonstrieren.

Duck-Typing objektorientiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Reflection in Java: import java.lang.reflect.*;

```
public static int demonstrateDuckTyping ( Object obj ) {
   Integer i = 123;
   String str = "Hello World";
   Class<?> c = obj.getClass();
    // On last slide: Class<?> c = Class.forName ( nameOfSomeClass );

   .....
}
```

Dafür ist im Grunde nur eine kleine Änderung am Beispiel eben notwendig. Wir bekommen jetzt nicht mehr den voll qualifizierten Namen einer Klasse als String, sondern ein Objekt von einem unbekannten Referenztyp. Mit getClass bekommen wir aber dessen Class-Objekt. Alles Weitere ist dann identisch: Wir holen uns eine Methode, die wir aufrufen wollen, aus diesem Class-Objekt, und rufen sie wie eben gesehen auf. Die dynamische Prüfung, ob die Methode existiert und ob alles so stimmt, erfolgt über die von getDeclaredMethod beziehungsweise invoke geworfenen Exceptions.

***Abschließende Nebenbemerkung*: Java Beans haben wir uns in der FOP nicht angeschaut, das ist ein fortgeschrittenes, vielfältig einsetzbares Konzept für Duck Typing in Java, das auf Reflection beruht.**