Kapitel 10

Polymorphie versus Typsicherheit

Skript zur Vorlesung
Einführung in die Programmierung
im Wintersemester 2012/13
Ludwig-Maximilians-Universität München

Übersicht

Übersicht

10.1 Typvariablen

- Wir haben das Konzept der *Polymorphie* im Zusammenhang mit Vererbung kennengelernt: Eine Methode, die Objekte der Klasse `A` als Parameter bekommen kann, kann auch Objekte jeder Unterklasse von `A` als Parameter bekommen.
- Manchmal kann es wünschenswert sein, eine Methode (oder Klasse) zwar allgemein zu definieren, beim Verwenden aber sicher zu sein, dass sie jeweils nur mit einem ganz bestimmten Typ verwendet wird.
- Lösung: Man führt eine *Typvariable* ein.

10.1 Typvariablen

- Variablen, die wir in Programmen bisher gesehen haben, sind Variablen, die einen speziellen Typ (primitiver oder Objekt-Typ) haben und verschiedene Werte dieses Typs annehmen können, aber nicht verschiedene Typen (außer vererbungs-polymorphe Typen).
- Eine *Typvariable* hat als Wert einen Typ.
- Der Typ einer Typvariablen ist „der Typ aller Typen“, in Java auch `class` genannt.

	Typ	Wert
Variable	ein beliebiger, aber fester Typ `A`	ein beliebiger Wert des Typs (z.B. auch ein Objekt der Klasse) `A` (oder einer Unterklasse)
Typvariable	**`class`**	eine beliebige Klasse

10.1 Typvariablen

- Das Konzept der Typvariablen haben wir bereits kennengelernt:
  - In der Definition von Eigenschaften zweistelliger Relationen wurde eine Typvariable verwendet

    $$R \subseteq M \times M$$

    Hier ist M eine Typvariable und steht für eine beliebige Menge. Damit ist R Teilmenge des zweistelligen Kreuzproduktes *derselben* Menge M.
  - Eigenschaften von Funktionen wurden abstrakt für die Menge D als Definitionsbereich und die Menge B als Bildbereich definiert – D und B sind wiederum Typvariablen.
  - Wiederum mit Typvariablen wurden die Signaturen verschiedener Funktionen angegeben, z.B.:
    - Projektion auf Folgen: $\pi : M^n \times I_n \rightarrow M$
    - Postfix: $postfix : M^* \times M \rightarrow M$
  - Auch Sorten von Ausdrücken wurden mit Variablen bezeichnet, diese Variablen können wir auch als Typvariablen auffassen.

10.1 Typvariablen

- Nicht nur Polymorphie, auch Überladung haben wir mit Typvariablen charakterisiert:
  - Vorzeichenoperator: $S \to S$ mit $S \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \ldots\}$
  - Arithmetische Operatoren: $S \times S \to S$ mit $S \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \ldots\}$
  - Vergleichsoperatoren: $S \times S \to \texttt{boolean}$ mit $S \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \ldots\}$
  - Type-Cast-Operatoren hätten wir auch so angeben können:
    $(\texttt{int}) : A \to \texttt{int}$ mit $A \in \{\texttt{char}, \texttt{byte}, \texttt{short}, \ldots\}$

Fußnote:

Typ-Casting hatten wir bisher für primitive Typen kennen gelernt. Auch für Objekttypen gibt es analog entsprechende Typ-Cast-Operatoren: Für eine Klasse `K` ist dies der Operator `(K)`: `O` → `K`

wobei `O` nur eine Klasse (Objekttyp) sein darf, die mit K eine *Verebungsbeziehung* hat (auch über mehrere Stufen). ***In welche Richtung gibt es einen Informationsverlust?***

10.1 Typvariablen

- In all diesen Fällen beschreiben wir mit Hilfe der Typvariablen die abstrakte Syntax. Die Syntax legt bereits fest, dass mit jedem Vorkommen derselben Typvariablen *S* der selbe Typ bezeichnet wird.
- Für die Semantik (Bedeutung) benötigen wir wiederum eine Variablen-Substitution, also eine Belegung der Typvariable mit einem konkreten Wert (d.h., einem *bestimmten* Typ).

Übersicht

10.2 Polymorphie: Revisited

10.2.1 Ploymorphie bei Methoden

- Eine Art Polymorphie ist das *Überladen* von Methoden (gleiche Namen aber unterschiedlichen Parameter-Typen):

```
public static void methode(Object o)
{
        System.out.println("01");
}

public static void methode(Tier t)
{
        System.out.println("02");
}

public static void methode(Schaf s)
{
        System.out.println("03");
}
```

Diese drei Methoden sind unterschiedlich!

10.2 Polymorphie: Revisited

10.2.1 Ploymorphie bei Methoden

- Inferenz der richtigen Methode bei überladenen Methoden:

  Anhand des (*Laufzeit-*)Typs eines Parameters kann die Laufzeitumgebung entscheiden, welche Methode aufgerufen wird:

```
Schaf s = new Schaf();
Object o = (Object) s;
Tier t = (Tier) s;
methode(o);
methode(t);
methode(s);
```

Ausgabe:

```
01
02
03
```

10.2 Polymorphie: Revisited

10.2.1 Ploymorphie bei Methoden

- Eine andere Art der Polymorphie ist das *Überschreiben* von *Objekt-Methoden* in Unterklassen. Hier wird der tatsächliche Typ des Objektes bestimmt und die entsprechende Methode verwendet:

```
public class Tier
{
      public String toString()
      {
                return "Tier";
      }
}
```

```
public class Schaf extends Tier
{
     public String toString()
     {
          return "Schaf";
     }
 }
```

```
Schaf s = new Schaf();
Object o = (Object) s;
Tier t = (Tier) s;
System.out.println(o.toString()); // Ausgabe: Schaf
System.out.println(t.toString()); // Ausgabe: Schaf
System.out.println(s.toString()); // Ausgabe: Schaf
```

10.2 Polymorphie: Revisited

10.2.1 Ploymorphie bei Methoden

- Bei überladenen Methoden entscheidet die Laufzeitumgebung für eine gegebene aktuelle Parametrisierung, welche Methode für diese Parametrisierung die *spezifischste* ist. Diese wird ausgeführt.

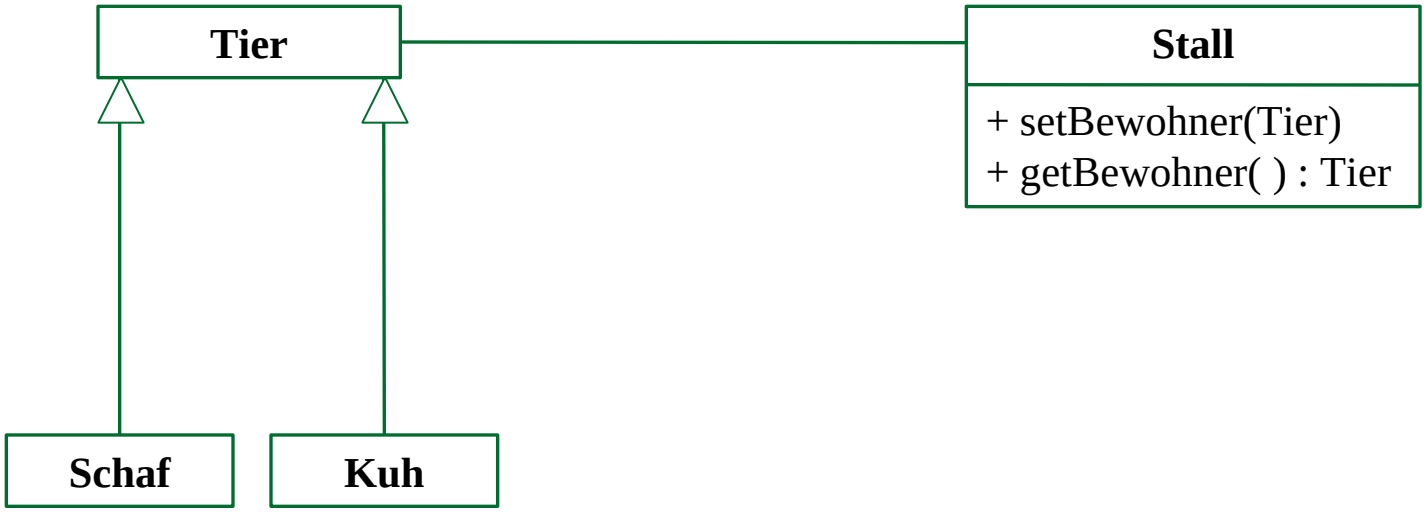
```
public static void methode(Object o)
{
        System.out.println("01");
}

public static void methode(Tier t)
{
        System.out.println("02");
}

public static void methode(Schaf s)
{
        System.out.println("03");
}

Schaf s = new Schaf();
methode(s); // Ausgabe: 03
```

Obwohl der Parameter `s` nicht nur vom Typ `Schaf`, sondern (aufgrund der Vererbung) auch vom Typ `Tier` und `Object` ist, wird die dritte Methode verwendet, die spezifischste für den aktuellen Parameter.

10.2 Polymorphie: Revisited

10.2.2 Typ-Polymorphie

- Wie wir gesehen haben, ermöglicht Vererbung Polymorphie.
- Gegeben folgendes Beispiel:



10.2 Polymorphie: Revisited

10.2.2 Typ-Polymorphie

- Eine Klasse `Stall` kann in diesem Beispiel für ein Objekt vom Typ `Tier` entworfen werden:

```
public class Tier
{
}
public class Stall
{
  private Tier bewohner;

  public void setBewohner(Tier tier)
  {
    this.bewohner = tier;
  }

  public Tier getBewohner()
  {
    return this.bewohner;
  }
}
```

- Der Stall kann jedes beliebige Objekt, das vom Typ `Tier` oder vom Typ einer Unterklasse von `Tier` ist, aufnehmen:

```
public class Schaf extends Tier
{
}
public static void main(String[] args)
{
  Stall schafstall = new Stall();
  schafstall.setBewohner(new Schaf());
}
```

10.2 Polymorphie: Revisited

10.2.2 Typ-Polymorphie

- *Problem:*
  Wenn man das Tier wieder aus dem Stall führt, "weiß" der Stall nicht mehr, was für ein Tier es ist:

```
public static void main(String[] args)
{
  Stall schafstall = new Stall();
  schafstall.setBewohner(new Schaf());

  Tier bewohner = schafstall.getBewohner();
}
```

- *Lösung ohne generische Typen:*
  Der Programmierer merkt es sich und führt einen expliziten Type-Cast durch:

```
public static void main(String[] args)
{
  Stall schafstall = new Stall();
  schafstall.setBewohner(new Schaf());

  Tier bewohner = schafstall.getBewohner();
  Schaf schaf = (Schaf) schafstall.getBewohner();
}
```

10.2 Polymorphie: Revisited

10.2.2 Typ-Polymorphie

- Die oben skizzierte "klassische" Lösung ist bestenfalls lästig für den Programmierer (da häufig explizite Type-Casts nötig sind).
- In größeren Anwendungen (mit vielen Programmierern, die am gleichen Projekt arbeiten) ist diese Lösung auch überaus fehleranfällig und daher gefährlich (und bestenfalls teuer), da die Typüberprüfung erst zur *Laufzeit* stattfindet:

```
public class Kuh extends Tier
{
}
...
  // Code Fragment
  Stall stall = new Stall();
  stall.setBewohner(new Schaf());

  // Code an entfernter Stelle (anderer Programmierer)
  stall.setBewohner(new Kuh());

  // wiederum andere Stelle (gueltiger Code!)
  Schaf schaf = (Schaf) stall.getBewohner(); // RuntimeException:
                                             // ClassCastException
```

Übersicht

10.3 Generische Klassen

10.3.1 Intuition

- In diesem Kapitel lernen wir eine neuere Lösung kennen, die mit Version 5.0 in Java eingeführt wurde: Typ-Polymorphie wird nicht nur durch Vererbung, sondern auch durch *generische Klassen* ermöglicht (*Generics*). Das ist nicht nur bequemer, sondern ermöglicht die Typüberprüfung normalerweise bereits zur *Übersetzungszeit* (und nicht erst zur Laufzeit, wo es dann einen Fehler geben kann)
- Eine generische Klasse ist *allgemein* für variable Typen (mit Hilfe von *Typvariablen*) implementiert – die Typen können bei der Verwendung der Klasse festgelegt werden (entspricht der Substitution der Typvariablen mit konkreten Typen), ohne dass die Implementierung verändert werden muss.
- Beide Konzepte der Typ-Polymorphie können auch gleichzeitig verwendet werden.

10.3 Generische Klassen

10.3.1 Intuition

- Beispiel: Die Klasse `Stall` wird durch Einführung einer *Typvariablen* generisch gehalten
- Im Gebrauch wird die Klasse durch Belegen der Typvariable mit einem bestimmten *parametrisierten Typ*

```
public class Stall<T> // Typvariable: T
{
  private T bewohner;

  public void setBewohner(T tier)
  {
    this.bewohner = tier;
  }

  public T getBewohner()
  {
    return this.bewohner;
  }
}
```

Typvariable `T` wird mit `Schaf` belegt. Damit muss überall, wo `T` stand nun `Schaf` stehen.

Kein Casting mehr nötig, da Methode `getBewohner()` den Ergebnistyp `T` also `Schaf` hat!

```
// Code Fragment
Stall<Schaf> stall = new Stall<Schaf>();
stall.setBewohner(new Schaf());

// dieses Fragment ist nicht mehr moeglich
// Kompilierfehler:
// The method setBewohner(Schaf) in the type Stall<Schaf>
// is not applicable for the arguments (Kuh)
stall.setBewohner(new Kuh());

// wiederum andere Stelle (gueltiger Code!)
Schaf schaf = stall.getBewohner();
// keine RuntimeException moeglich
```

10.3 Generische Klassen

10.3.1 Intuition

- Die Einführung einer *Typvariablen* bei der Definition der generischen Klasse `Stall` erinnert ein wenig an Parameter bei Methoden
- Im folgenden Programm `Streckenberechnung` kommen Parameter in `strecke` vor als
  - *formale* Parameter
    `m`, `t`, `k` (in Zeile 2)
  - *aktuelle* Parameter
    `2.3`, `42`, `17.5` (in Zeile 10)

```
1   public class Streckenberechnung {
2     public static double strecke(double m, double t, double k)
3     {
4       double b = k / m;
5       return 0.5 * b * (t * t);
6     }
7
8     public static void main(String[] args)
9     {
10      System.out.println(strecke(2.3, 42, 17.5));
11    }
12  }
```

- Die Berechnungsvorschrift wird abstrakt mit den formalen Parametern definiert, die im Methodenrumpf verwendet werden.
- Die konkrete Berechnung wird aber mit Werten durchgeführt, die den Parametern zugewiesen werden (den aktuellen Parametern), d.h. die formalen Parameter werden durch diese Werte substituiert.

10.3 Generische Klassen

10.3.1 Intuition

Analog bei Typvariablen:

- Der Typparameter bei Definition der Klasse ein *formaler* Typparameter
- Konkreter Gebrauch der Klasse durch Belegen der Typvariable mit einem bestimmten Typ (*aktuellen* Typparameter); Typprüfung dadurch möglich:

```
public class Stall<T> // Typvariable: T (formaler Typ-Parameter)
{
  private T bewohner;

  public void setBewohner(T tier)
  {
    this.bewohner = tier;
  }

  public T getBewohner()
  {
    return this.bewohner;
  }
}
```

```
// Code Fragment
Stall<Schaf> stall = new Stall<Schaf>(); // aktueller Typ-Parameter: Schaf
                                         // ermoeglicht Typueberpruefung zur
                                         // Uebersetzungszeit
stall.setBewohner(new Schaf());

// dieses Fragment ist nicht mehr moeglich
// Kompilierfehler:
// The method setBewohner(Schaf) in the type Stall<Schaf>
// is not applicable for the arguments (Kuh)
stall.setBewohner(new Kuh());

// wiederum andere Stelle (gueltiger Code!)
Schaf schaf = stall.getBewohner();
// keine RuntimeException moeglich
```

10.3 Generische Klassen

10.3.2 Umsetzung

- Deklaration einer generischen Klasse:

  ```
  class <Klassenname><<Typvariable(n)>>
  ```

  wenn mehr als eine Typvariable in `<Typvariable(n)>` definiert wird, werden die einzelnen Variablen durch Komma getrennt
- *Typisierung* der Klasse (Parametrisierung des Typs):

  ```
  <Klassenname><<Typausdruck/Typausdruecke>>
  ```
- Beispiele für Instantiierungen von `Stall<T>`:
  - `Stall<Tier>`
  - `Stall<Schaf>`
  - `Stall<Kuh>`
  - `Stall<Schaf,Kuh>` *`//ungueltig: zu viele Parameter`*
  - `Stall<String>` *`//gueltig, aber vermutlich unerwuenscht`*
  - `Stall<int>` *`//ungueltig (Warum???)`*

10.3 Generische Klassen

10.3.2 Umsetzung

- Eine generische Klasse kann also auch mit mehreren Typ-Parametern definiert werden:

```
public class TripelStall<S, T, U> {
            private S ersterBewohner;
            private T zweiterBewohner;
            private U dritterBewohner;
            ...
}
```

- Beispiele für Instantiierungen von `TripleStall<S, T, U>`:
  - `Stall<Tier, Tier, Tier>`
  - `Stall<Tier, Schaf, Kuh>`
  - `Stall<Kuh, Schaf, Kuh>`
  - `Stall<Schaf, Kuh, Kuh, Kuh>` *// ungueltig: zu viele Parameter*

10.3 Generische Klassen

10.3.2 Umsetzung

- Zur *Übersetzung* von generischen Typen gibt es grundsätzlich zwei Möglichkeiten:
  1. *Heterogene Übersetzung*: Für jede Instantiierung (`Stall<Tier>`, `Stall<Schaf>`, `Stall<Kuh>` etc.) wird individueller Byte-Code erzeugt, also drei unterschiedliche (heterogene) Klassen.
  2. *Homogene Übersetzung*: Für jede parametrisierte Klasse (`Stall<T>`) wird genau eine Klasse erzeugt, die die generische Typinformation löscht (*type erasure*) und die Typ-Parameter durch die Klasse `Object` ersetzt. Für jeden konkreten Typ werden zusätzlich Typanpassungen in die Anweisungen eingebaut.
- Java nutzt die homogene Übersetzung (C++ die heterogene).
- Der Byte-Code für eine generische Klasse entspricht also in etwa dem Byte-Code einer Klasse, die nur Typ-Polymorphie durch Vererbung benutzt.
- Gewonnen hat man aber die Typ-Sicherheit zur Übersetzungszeit.

10.3 Generische Klassen

10.3.2 Umsetzung

- Generische Klassen können auch ohne Typ-Parameter verwendet werden. (Genannt: *Raw-Type*)
- Raw-Types bieten die gleiche Funktionalität wie parametrisierte Typen, allerdings werden die Parametertypen nicht zur Übersetzungszeit überprüft.
- Beispiel: Raw-Type von `Stall<T>` ist Stall.

```
Stall<Schaf> schafstall = new Stall<Schaf>();
Stall stall = new Stall();
stall = schafstall;

stall.setBewohner(new Kuh()); // Warnung:
              // Type safety: The method setBewohner(Tier)
              // belongs to the raw type Stall.
              // References to generic type Stall<T>
              // should be parameterized
// Die Warnung ist gerechtfertigt, denn:
Schaf poldi = schafstall.getBewohner();
// ist gueltiger Code, der aber zu einer
// RuntimeException (ClassCastException) fuehrt
```

10.3 Generische Klassen

10.3.3 Vererbung und Typ-Schranken

- Von generischen Klassen lassen sich in gewohnter Weise Unterklassen ableiten:

```
public class SchafStall extends Stall<Schaf>
{
}
```

- Dabei kann der Typ der Oberklasse weiter festgelegt werden (hier wird die Oberklasse typisiert – andere Möglichkeiten sehen wir später).
- Die Unterklasse einer generischen Klasse kann auch selbst wieder generisch sein.

```
public class GrossviehStall<T> extends Stall<T>
{
}
```

- Im nächsten Beispiel wird sogar ein weiterer Typ eingeführt:

```
public class DoppelStall<T, S> extends Stall<T>
{
        private S zweiterBewohner;
        ...
}
```

- Wie kann die unerwünschte Typisierung `Stall<String>` von `Stall<T>` verhindert werden?
- Die Typvariable kann innerhalb einer (Vererbungs-)Typ-Hierarchie verortet werden, indem eine *obere Schranke* angegeben wird:

```
public class Stall<T extends Tier>
{
    ...
}
    // Code Fragment
    Stall<String> // ungueltig:
                  // String ist nicht Unterklasse von Tier
```

10.3 Generische Klassen

10.3.3 Vererbung und Typ-Schranken

- Analog zur oberen Schranke kann man auch eine *untere Schranke* definieren.
- Das ist jedoch nur für sog. Wildcards (als `?` dargestellt) möglich, und nicht in Klassen-Definitionen, sondern *nur in Variablen-Deklarationen*:

```
Stall<? super Schaf> stall;
...
// Code Fragment
stall.setBewohner(new Kuh()) // ungueltig:
        // The method setBewohner(capture-of ? super Schaf)
        // in the type Stall<capture-of ? super Schaf> is
        // not applicable for the arguments (Kuh)
```

10.3 Generische Klassen

10.3.3 Vererbung und Typ-Schranken

- Wann ist eine obere Schranke und wann eine untere Schranke sinnvoll?
  - Wenn der Typ-Parameter `T` nur als Argument von Objekt-Methoden verwendet wird, ist oft eine untere Schranke sinnvoll (`? super T`).
  - Wenn der Typ-Parameter `T` nur bei Rückgabewerten von Objektmethoden eine Rolle spielt, kann man dem Benutzer oft mehr Freiheit geben, indem man eine obere Schranke benützt (`? extends T`).
- In der Unterklasse einer generischen Klasse können neue Schranken eingeführt werden.
- Mit dem Token `&` können dabei mehrere Schranken verbunden werden.

```
public interface Grossvieh
{ … }
public class GrossviehStall<T extends Tier & Grossvieh> extends Stall<T>
{ … }
```

- Ab der zweiten oberen Schranke kann es sich dabei natürlich nur noch um Interfaces handeln. **Warum?**

10.3 Generische Klassen

10.3.4 Generische Klassen in UML

- In UML heißen typisierte Klassen auch *Templates*.
- Unser Beispiel wird mit Templates in UML so dargestellt: