

Skript zur Vorlesung:
**Einführung in die
Programmierung**
WiSe 2009 / 2010

Skript © 2009 Christian Böhm, Peer Kröger, Arthur Zimek

Prof. Dr. Christian Böhm
Annahita Oswald
Bianca Wackersreuther

Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme



16. Typisierte Klassen

16.1 Grundlagen

16.2 Wiederholung: Polymorphismus vs. Überladung

16.3 Grenzen der Typ-Polymorphie durch Vererbung

16.4 Typvariablen und generische / typisierte Klassen

16.5 Vererbung bei typisierten Klassen

16.6 Wildcards, obere und untere Typ-Schranken

16.7 Generische Methoden

16.8 Ober- und Unterklassen bei typisierten Klassen

16.9 Typisierte Klassen in UML

- 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.

- Variablen, die wir in Programmen bisher gesehen haben, sind Variablen, die verschiedene Werte annehmen können, aber nicht verschiedene Typen (außer vererbungs-polymorphe Typen).
- Eine *Typvariable* hat als Wert einen Typ.
- Der Typ einer Typvariablen ist immer der gleiche:
Der Typ aller Typen.
- **Was ist dieser generelle Typ in Java?**

Variablen vs. Typvariablen

	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

- Zur Erinnerung: In der Definition von Eigenschaften zweistelliger Relationen wurde eine Typvariable verwendet

$$R \subseteq M \times M$$

Hier steht M für eine beliebige Menge. Wichtig ist aber, dass R Teilmenge des zweistelligen Kreuzproduktes *derselben* Menge ist.

- 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.

- Nicht nur Polymorphie, auch Überladung haben wir mit Typvariablen charakterisiert:
 - Vorzeichenoperator: $S \rightarrow S$ mit $S \in \{\mathbf{byte}, \mathbf{short}, \mathbf{int}, \dots\}$
 - Arithmetische Operatoren: $S \times S \rightarrow S$ mit $S \in \{\mathbf{byte}, \mathbf{short}, \mathbf{int}, \dots\}$
 - Vergleichsoperatoren: $S \times S \rightarrow \mathbf{boolean}$
mit $S \in \{\mathbf{byte}, \mathbf{short}, \mathbf{int}, \dots\}$
 - Type-Cast-Operatoren hätten wir auch so angeben können:
 $(\mathbf{int}) : A \rightarrow \mathbf{int}$ mit $A \in \{\mathbf{char}, \mathbf{byte}, \mathbf{short}, \dots\}$
- 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.
- Die Semantik (Bedeutung) kommt zustande durch Belegung der Typvariable mit einem konkreten Wert (d.h., einem *bestimmten* Typ).

Wir haben gesehen, dass Methoden den gleichen Namen haben können, wenn sich die Parameter-Typen unterscheiden.

```

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, obwohl sie den gleichen Namen haben. Sie sind *überladen*.

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
```

Abgrenzung: Polymorphie von Methoden

Die Polymorphie von *Objekt-Methoden* durch Überschreiben in Unterklassen ist dagegen ein anderes Phänomen. 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
```

Bei überladenen Methoden entscheidet die Laufzeitumgebung für eine gegebene aktuelle Parametrisierung, welche Methode für diese Parametrisierung die spezifischste ist. Die spezifischste Methode 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.



Typ-Polymorphie durch Vererbung

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



- Eine Klasse `Stall` wurde in diesem Beispiel für Objekte vom Typ `Tier` entworfen.

```

public class Tier
{
}

public class Stall
{
    private Tier bewohner;

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

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

```

- **Welche Probleme können bei dieser Modellierung auftreten?**

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());
}
```

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 Typecast durch:

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

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

- Die oben skizzierte “klassische” Lösung ist bestenfalls lästig für den Programmierer (da häufig explizite Typecasts 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
    
```

- In diesem Kapitel lernen wir eine neuere Lösung kennen, die mit Version 5.0 in Java eingeführt wurde: 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*.
- Eine generische Klasse ist *allgemein* für variable Typen implementiert – die Typen können bei der Verwendung der Klasse festgelegt werden, ohne dass die Implementierung verändert werden muss.
- Beide Konzepte der Polymorphie können auch gleichzeitig verwendet werden.

Beispiel: Die Klasse `Stall` wird durch Einführung einer Typvariablen generisch gehalten:

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

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

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

Im Gebrauch wird die Klasse durch Belegen der Typvariable mit einem bestimmten Typ typisiert (man spricht dann von einem parametrisierten Typ):

```
// 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
```

```

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 }

```

- Im Programmbeispiel kommen Parameter der Methode `strecke` in zwei Varianten vor:
 1. Als *formale* Parameter `m`, `t`, `k` (Zeile 2)
 2. Als *aktuelle* Parameter `2.3`, `42`, `17.5` (Zeile 10)
- 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, eben den aktuellen Parametern.

Entsprechend ist der Typparameter bei Definition der Klasse ein *formaler* Typparameter:

```
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;
    }
}
```

Im Gebrauch der Klasse und Belegen der Typvariable mit einem bestimmten Typ kennt der Compiler dann den *aktuellen* Typparameter und kann dadurch eine Typprüfung durchführen:

```
// Code Fragment
Stall<Schaf> stall = new Stall<Schaf>(); // aktueller Typ-Parameter: Schaf
                                         // ermöglicht 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
```

- Deklaration einer generischen Klasse:

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

- Instantiierung eines generischen Typs (=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 unerwuenscht*
- `Stall<int>` *//ungueltig (Warum???)*

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.
- Vorteil / Nachteil der homogenen Übersetzung:



Weniger erzeugter Byte-Code, Rückwärtskompatibilität.



Geringere Ausdrucksmächtigkeit der Typüberprüfung zur Übersetzungszeit.

- 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

```

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).

Unterklassen von generischen Klassen

- Die Unterklasse einer generischen Klasse kann auch selbst wieder generisch sein.

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

- In diesem Beispiel wird sogar ein weiterer Typ eingeführt:

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

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

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

- Zur Typisierung kann man auch Wildcards verwenden:

```
public class IrgendeinStall<?>
```

- Das ist so zunächst nur sinnvoll, wenn der genaue Typ keine Rolle spielt.
- Sinnvolle Verwendung finden Platzhalter zusammen mit oberen und / oder unteren Schranken.

- **Wie kann die unerwünschte Typisierung `Stall<String>` verhindert werden?**
- Die Typvariable kann innerhalb einer 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
```

Analog zur oberen Schranke kann man auch eine untere Schranke definieren. Das ist jedoch nur für Wildcards möglich, und nicht in Klassen-Definitionen, sondern nur in Deklarationen von Variablen:

```

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)

```

Die untere Schranke wird in der Praxis wesentlich seltener verwendet als die obere Schranke. Wir wollen aber den Sinn der unteren Schranke an einem einfachen Beispiel klar machen:

Bauer Moosgruber möchte einen Wettbewerb “Schönster Schafstall” veranstalten. Dazu bastelt er eine Klasse, die den Wettbewerb anhand eines übergebenen Vergleichsalgorithmus durchführt:

```
public interface Vergleich<T>
{
    public int vergleiche(T t1, T t2);
}
public class SchoensterSchafstall
{
    public SchoensterSchafstall(Vergleich<Stall<Schaf>> vergleich)
    {
        // ...
    }
}
```

Gehen Sie nun davon aus, dass Bauer Huber im letzten Jahr einen Wettbewerb “Schönster Stall” durchgeführt und dafür einen Vergleichsalgorithmus entworfen hat:

```
public class StallVergleicher implements Vergleichier<Stall<Tier>>
{
    public int vergleiche(Stall<Tier> t1, Stall<Tier> t2)
    {
        ...
    }
}
```

Bauer Moosgruber würde nun für seinen Wettbewerb gerne Hubers Klasse `StallVergleicher` verwenden. Hubers `Vergleicher` passt aber nicht zur Parametrisierung von Moosgrubers Konstruktor:

```
public SchoensterSchafstall(Vergleicher<Stall<Schaf>> vergleicher)
```

Wie kann Moosgruber seinen Wettbewerb so definieren, dass er Hubers `StallVergleicher` verwenden kann?

Moosgruber muss die unnötige Restriktion von seinem Konstruktor aufweichen:

```
public class SchoensterSchafstall
{
    public SchoensterSchafstall(Vergleicher<Stall<? super Schaf>> vergleicher)
    {
        ...
    }
}
```

Ein Vergleicher, der Ställe aller möglichen Oberklassen von Schaf vergleichen kann, kann natürlich auch Schaf-Ställe vergleichen – aber nicht umgekehrt.

- Wenn der Typ-Parameter \mathbb{T} nur als Argument verwendet wird, ist oft eine untere Schranke sinnvoll (? **super** \mathbb{T}).
- Wenn der Typ-Parameter \mathbb{T} nur bei Rückgabewerten eine Rolle spielt, kann man dem Benutzer oft mehr Freiheit geben, indem man eine obere Schranke benützt (? **extends** \mathbb{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?**

- Betrachten wir nun eine statische Methode, um ein Tier in den Stall zu treiben: Der Stall ist dabei zunächst scheinbar ein Stall für ein beliebiges Objekt.

```
public class Stall<T extends Tier>
{
    // statische Methode
    public static void treibeInDenStall(Tier tier, Stall<?> stall)
    {
        stall.setBewohner(tier);
    }
}
```

- **Geht das?**

- Der Wildcard bedeutet, wir haben einen Stall von unbekanntem Typ (“Stall of unknown”).
- `Tier` ist aber ein ganz bestimmter Typ.
- Der Compiler kann nicht entscheiden, ob `Tier` (oder irgendein anderer Typ) der gleiche Typ ist wie der mit dem Wildcard bezeichnete.
- Deshalb gibt es einen Fehler zur Übersetzungszeit:

```
// The method setBewohner(capture-of ?)
// in the type Stall<capture-of ?>
// is not applicable for the arguments (Tier)
```

- Wildcards sind also nur in Verbindung mit Schranken sinnvoll oder wenn der Typ tatsächlich irrelevant (oder unbekannt) ist.

- Die obige Methode lässt sich dennoch generisch typisieren:

```
public static <T extends Tier> void treibeInDenStall(T tier, Stall<T> stall)
{
    stall.setBewohner(tier);
}
```

- Man definiert einen Typ-Parameter spezifisch für diese Methode. Die Definition wird vor dem Rückgabetyt der Methode eingeführt.
- Achtung: Auch falls diese Methode in der Klasse `class Stall<T extends Tier>` definiert wird, ist der Typ-Parameter `T` in der Methode nicht identisch mit dem Typ-Parameter der Klasse, da letzterer nicht-statisch, der Typ-Parameter der (statischen) Methode aber eine statische Eigenschaft ist. Als solche ist sie unabhängig von jeder möglichen Instantiierung der Klasse!

- Wenn ein Parameter ein generischer Typ ist, der genaue Typ aber irrelevant ist, kann man anstelle einer generischen Methode auch einfach eine Methode mit Platzhalter definieren:

```
public static boolean stallBesetzt(Stall<?> stall)
{
    return stall.getBewohner() != null;
}
```

- Dies ist dann möglich, wenn:
 - Der Typ-Parameter T nur einmal benutzt wird,
 - der Typ des Rückgabewertes nicht von T abhängt,
 - der Typ keines anderen Argumentes der Methode von T abhängt,
 - das entsprechende Argument für Typ-Polymorphie verwendet wird (d.h., es sind verschiedene Argumenttypen erlaubt).
- Generische Methoden hingegen werden verwendet, um Abhängigkeiten zwischen mehreren Argumenttypen oder Argumenttypen und Rückgabetyper einer Methode herzustellen.

- Wie wir aus dem Kapitel Vererbung wissen, ist folgende Zuweisung kein Problem:

```
Object objekt = new Object();  
Integer integer = new Integer(1);  
objekt = integer;
```

- **Wie sieht es in folgendem Fall aus?**

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

- Dieser Versuch ist ungültiger Code. Er würde wieder zu folgendem altbekanntem Problem führen:

```
Stall<Schaf> schafstall = new Stall<Schaf>();
Stall<Tier> stall = new Stall<Tier>();
stall = schafstall; // ungueltig (Typfehler)
stall.setBewohner(new Tier());
Schaf poldi = schafstall.getBewohner(); // ClassCastException
```

- Der Grund dafür ist, dass zwar `Schaf` eine Unterklasse von `Tier`, aber `Stall<Schaf>` *nicht* Unterklasse von `Stall<Tier>` ist.

Mit der echten Unterklasse SchafStall **extends** Stall<Schaf> haben wir das Problem nicht. Hier kann eben auch nicht *irgendein* Tier angesiedelt werden, sondern nur ein Schaf.

```

SchafStall schafstall = new SchafStall();
Stall<Schaf> stall = new Stall<Schaf>();
stall = schafstall; // ok
stall.setBewohner(new Schaf());
// nicht moeglich:
// stall.setBewohner(new Tier());
Schaf poldi = schafstall.getBewohner();
  
```

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

