Überblick

9. Weitere Konzepte der oo Programmierung in Java
9.1 Vererbung, abstrakte Klassen, Polymorphismus
9.2 Interfaces
9.3 Ausnahmen
9.4 Polymorphie versus Typsicherheit

Grundlegendes zu Typsystemen

- Was würde passieren, wenn wir für zwei Objekte vom Typ `String` den Operator `&&` ausführen würden?
- In Java verhindert der Compiler, dass dieser Code überhaupt übersetzt wird ansonsten gäbe es zur Laufzeit einen Fehler (was schlecht wäre — Warum?).
- Um dies festzustellen, besitzt Java (wie die meisten höheren Programmiersprachen) ein *Typsystem*.

Grundlegendes zu Typsystemen

- Ein *Typsystem* ist eine Komponente einer Programmiersprache, die den Wertebereich von Variablen und den Bildbereich von Methoden einschränkt.
- Programmiersprachen mit Typsystem nennt man *typisiert*.
- Wesentliche Aufgabe ist es, durch Typisierung sicher zu stellen, dass auf den Inhalten von Variablen keine Operationen ausgeführt werden, die syntaktisch oder semantisch fehlerhaft sind (siehe unten).

Grundlegendes zu Typsystemen

Wichtige Bestandteile von Typsystemen sind:

- Die Typen selbst, die entweder mittels Typ-Definitionen (in Java: Klassen) erzeugt oder in der jeweiligen Sprache fest integriert sind (in Java: primitive Datentypen).
- Die Möglichkeit, Programmelemente (wie Variablen, Methodenparameter usw.) durch Deklaration an einen bestimmten Typ zu „binden“.
- Regeln, nach denen die Werte von Ausdrücken einem bestimmten Typ zugeordnet werden.
- Regeln zur Prüfung der Zuweisungskompatibilität von Typen.
- Optional weitere Sprachbestandteile wie typbezogene Operatoren (z.B. der `instanceof`-Operator in Java) zur Ermittlung und Prüfung von Typinformation zur Laufzeit.

Grundlegendes zu Typsystemen

Wesentliche Aufgaben von Typsystemen:

- Erkennung von Typverletzungen bei der Übersetzung und Ausführung:
  - Mit der Typinformation kann verhindert werden, dass Variablen Werte zugewiesen werden, welche die betreffende Variable niemals annehmen sollte (sogenannte Typinvarianten).
  - So wird die Ausführung von Operationen auf den Inhalten dieser Variablen, die entweder unmöglich oder nicht sinnvoll ist, verhindert.
- Typumwandlung (Type Conversion). In vielen Sprachen stehen hierfür explizite Operationen zur Verfügung.

Das Typsystem von Java

Eigenschaften des Typsystems von Java[20]:

- Starke Typisierung,
  d.h. die Typen werden streng unterschieden, Datentypen können umgewandelt werden, implizit aber nur ohne Informationsverlust; unsichere Typumwandlung ist zwar erlaubt, aber nur mit explizitem Typcast-Operator.
- Statische Typisierung,
  d.h. Typprüfung zur Übersetzungszeit (Achtung Ausnahme: Polymorphismus im Zusammenhang mit Vererbung!!!)
- Explizite Typisierung,
  d.h. Typen werden explizit benannt und nicht per „type inference“ abgeleitet.

[20] Die Kriterien sind nicht binär sondern eher fließend.

Das Typsystem von Java

- Tatsächlich ist Java daher „so gut wie“ *typsicher*, d.h. der Compiler kann Typverletzungen bereits zur Übersetzungszeit feststellen.
- Eine wichtige (und manchmal lästige Ausnahme) stellt die Polymorphie dar, wie wir in diesem Kapitel diskutieren werden.
- Das eigentlich sehr schöne Konzept der Polymorphie ist nicht generell „böse“, es hat nur bzgl. Typsicherheit eine Schwäche um die wir uns wie gesagt nun kümmern.

Polymorphie mit Typvariablen

- Wir haben das Konzept der *Polymorphie* im Zusammenhang mit Vererbung kennengelernt: Eine Methode, die Objekte der Klasse `A` als Parameter übergeben bekommen kann, kann auch Objekte jeder Unterklasse von `A` als Parameter übergeben 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 (d.h. typsicher ist).
- Lösung: Man führt eine *Typvariable* ein.

Polymorphie mit Typvariablen

- 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 *der Typ aller Typen*, in Java auch als **`class`** bezeichnet.
- Vergleich:

	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

Intuition von Typvariablen

- Das Konzept der Typvariablen kennen wir natürlich:
- 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.

Intuition von Typvariablen

- Auch das Überladen von Operatoren kann mit Typvariablen beschrieben werden, z.B.:
  - Arithmetische Operatoren: $S \times S \rightarrow S$ mit $S \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \ldots\}$
  - Vergleichsoperatoren: $S \times S \rightarrow \texttt{boolean}$ mit $S \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \ldots\}$
- 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).

Erinnerung: Überladen von 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!

Inferenz der richtigen Methode

- Anhand des (*Laufzeit-*)Typs eines Parameters kann JRE 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

- Eine andere Art der Polymorphie ist das *Überschreiben* von *Objekt-Methoden* in Unterklassen.
- Hier wird der tatsächliche Typ des Objektes (zur Laufzeit) 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
```

Überladene Methoden: Auswahl

- Bei überladenen Methoden entscheidet der Compiler für eine gegebene (deklarierte) aktuelle Parametrisierung welche Methode für diese Parametrisierung die *Spezifischste* ist.
- Dies ist ähnlich zur Auswahl der spezifischsten Objekt-Methode bei polymorphen Variablen.
- Diese spezifischste Methode wird jeweils ausgeführt.

Überladene Methoden: Auswahl

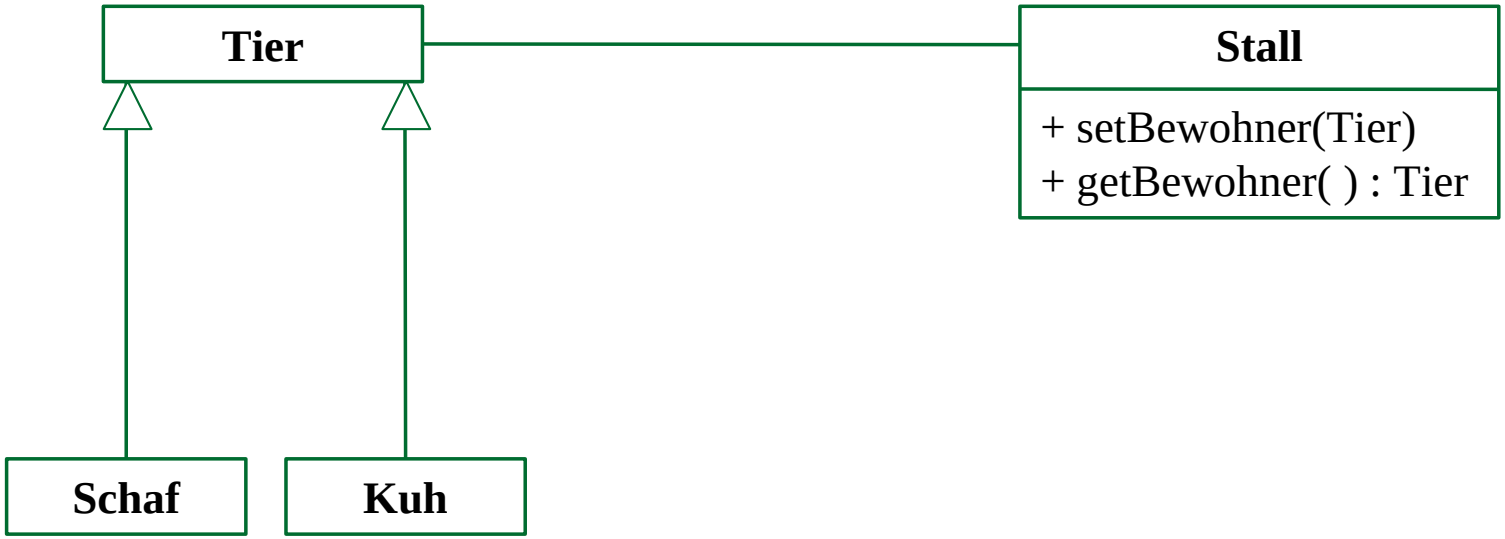
- Beispiel:

```
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:



Typ-Polymorphie durch Vererbung

- 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?

Typ-Polymorphie durch Vererbung

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());
}
```

Grenzen der Typ-Polymorphie durch Vererbung

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();
}
```

Grenzen der Typ-Polymorphie durch Vererbung

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());

    ...

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

Grenzen der Typ-Polymorphie durch Vererbung

- Das ist bestenfalls lästig für den Programmierer (häufige Typecasts).
- Das ist aber auch überaus fehleranfällig (v.a. bei vielen Programmiereren), 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
```

Typ-Polymorphie durch typisierte Klassen

- 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.

Generische Klassen

Beispiel: Die Klasse `Stall` wird durch Einführen 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;
  }
}
```

Typisierte Klassen

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

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

- Die Typvariable `T` wird hier für das Objekt `stall` mit `Schaf` substituiert, d.h. überall in der Definition von `Stall` steht beim Objekt `stall` nun `T` fuer `Schaf`.
- Folgendes Fragment ist also erlaubt:

```
stall.setBewohner(new Schaf());
```

Typisierte Klassen

- Das folgende Fragment ist aber nicht mehr erlaubt:

```
stall.setBewohner(new Kuh());
```

  Es führt zu einem Kompilierfehler:

```
// The method setBewohner(Schaf) in the type Stall<Schaf>
// is not applicable for the arguments (Kuh)
```

- Beim Herausnehmen (Getter) ist nun kein Cast mehr nötig, da der Rückgabetyp `T` also `Schaf` ist:

```
Schaf schaf = stall.getBewohner();
```

- Die Gefahr der Runtime-Exception wegen dem Typ-Cast ist gebannt und folgendes Konstrukt ist natürlich auch nicht mehr möglich:

```
Kuh muh = stall.getBewohner();
```

  denn auch das würde der Compiler nicht akzeptieren.

Erinnerung: Parameter bei Methoden

- Die Einführung einer Typvariablen bei der Definition der generischen Klasse Stall erinnert ein wenig an Parameter bei Methoden.

```
public class Streckenberechnung {
  public static double strecke(double m, double t, double k) {
    return 0.5 * k/m * (t * t);
  }

  public static void main(String[] args) {
    System.out.println(strecke(2.3, 42, 17.5));
  }
}
```

- Die Berechnungsvorschrift der Methode `strecke` wird abstrakt mit den formalen Parametern (`m`, `t`, `k`) definiert.
- 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.

Generische Klassen

- Entsprechend ist der Typparameter bei Definition der Klasse ein *formaler* Typparameter.

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

- Konkreter Gebrauch der Klasse durch Belegen der Typvariable mit einem bestimmten Typ (*aktuelle* Typparameter):

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

- Dadurch *Typsicherheit* (u.a. ist auch Typprüfung möglich):

```
Schaf schaf = stall.getBewohner();
```

Syntax für generische Typen

- 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???)*

Syntax für generische Typen

- 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<Kuh, Kuh, Kuh>`
  - `Stall<Tier, Schaf, Kuh>`
  - `Stall<Kuh, Schaf, Kuh>`
  - `Stall<Schaf, Kuh, Kuh, Tier>` *//ungueltig: zu viele Parameter*

Übersetzung von generischen Typen

Zur Übersetzung von generischen Typen gibt es grundsätzlich zwei Möglichkeiten:

- *Heterogene Übersetzung*: Für jede Instantiierung (`Stall<Tier>`, `Stall<Schaf>`, `Stall<Kuh>` etc.) wird individueller Byte-Code erzeugt, also drei unterschiedliche (heterogene) Klassen.
- *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.

Übersetzung von generischen Typen

- 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

Verwendung generischer Typen ohne Typ-Parameter

- Generische Klassen können auch ohne Typ-Parameter verwendet werden.
- Ein generischer Typ ohne Typangabe heißt *Raw-Type* und bietet die gleiche Funktionalität wie ein parametrisierter Typ, allerdings werden die Parametertypen nicht zur Übersetzungszeit überprüft.

Verwendung generischer Typen ohne Typ-Parameter

- 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
```

Unterklassen von generischen Klassen

- 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 (in diesem Beispiel wird die Oberklasse typisiert).

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

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

- Die Unterklasse kann auch einen weiteren Typ einführen:

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

Platzhalter (Wildcards)

- 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.

Obere Typ-Schranke

- Wie kann eine 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
```

Untere Typ-Schranke

- 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 Variable `stall` kann also nur Schafe oder deren Oberklassen aufnehmen (und damit keine Kühe).

Sinn von Schranken

- Wann ist eine obere, wann eine untere Schranke sinnvoll?
  - Wenn der Typ-Parameter `T` nur als Argument verwendet wird, ist oft eine untere Schranke sinnvoll (`? super T`).
  - Wenn der Typ-Parameter `T` nur bei Rückgabewerten 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.
- Ab der zweiten oberen Schranke kann es sich dabei natürlich nur noch um Interfaces handeln.