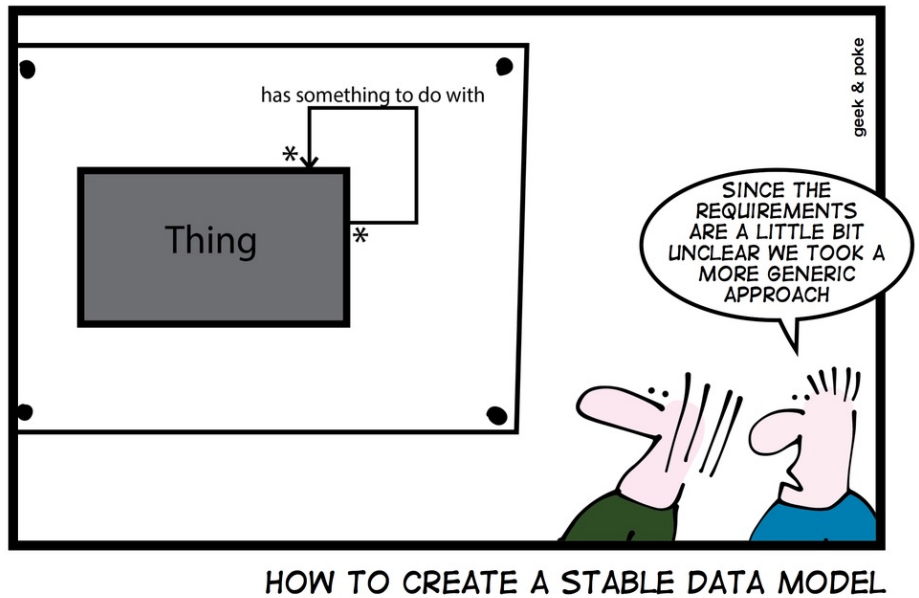
IO Streams & Generics

Vorlesungsinhalt



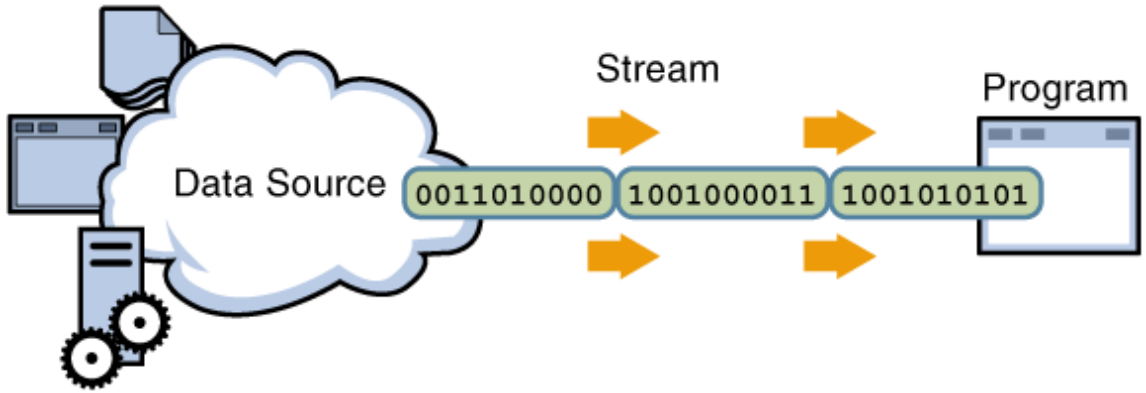
- I/O Streams
- Variable Argumentenanzahl
- Generics
- Aufgabe

I/O Streams

Was ist ein IO Stream? 1/2

- Bezeichnet in der Informatik eine kontinuierliche Abfolge von Datensätzen deren Ende nicht im Voraus abzusehen ist.
- Die Menge der Datensätze pro Zeiteinheit (Datenrate) kann variieren und so groß sein, dass nur begrenzte Ressourcen zur Weiterverarbeitung zur Verfügung stehen.
- Ein Stream kann von beliebigen Datentypen sein.
- Innerhalb eines Streams ändert sich der Datentyp nicht.

Was ist ein IO Stream? 2/2



- Datenströme können nicht als Ganzes, sondern nur fortlaufend verarbeitet werden.
- Zugriff auf einzelne Datensätze ist nur sequenziell möglich (im Gegensatz zu Datenstrukturen mit wahlfreiem Zugriff, wie z.B. Arrays)

Weitere Informationen unter:
https://javabeginners.de/Ein-_und_Ausgabe/Datenstroeme.php

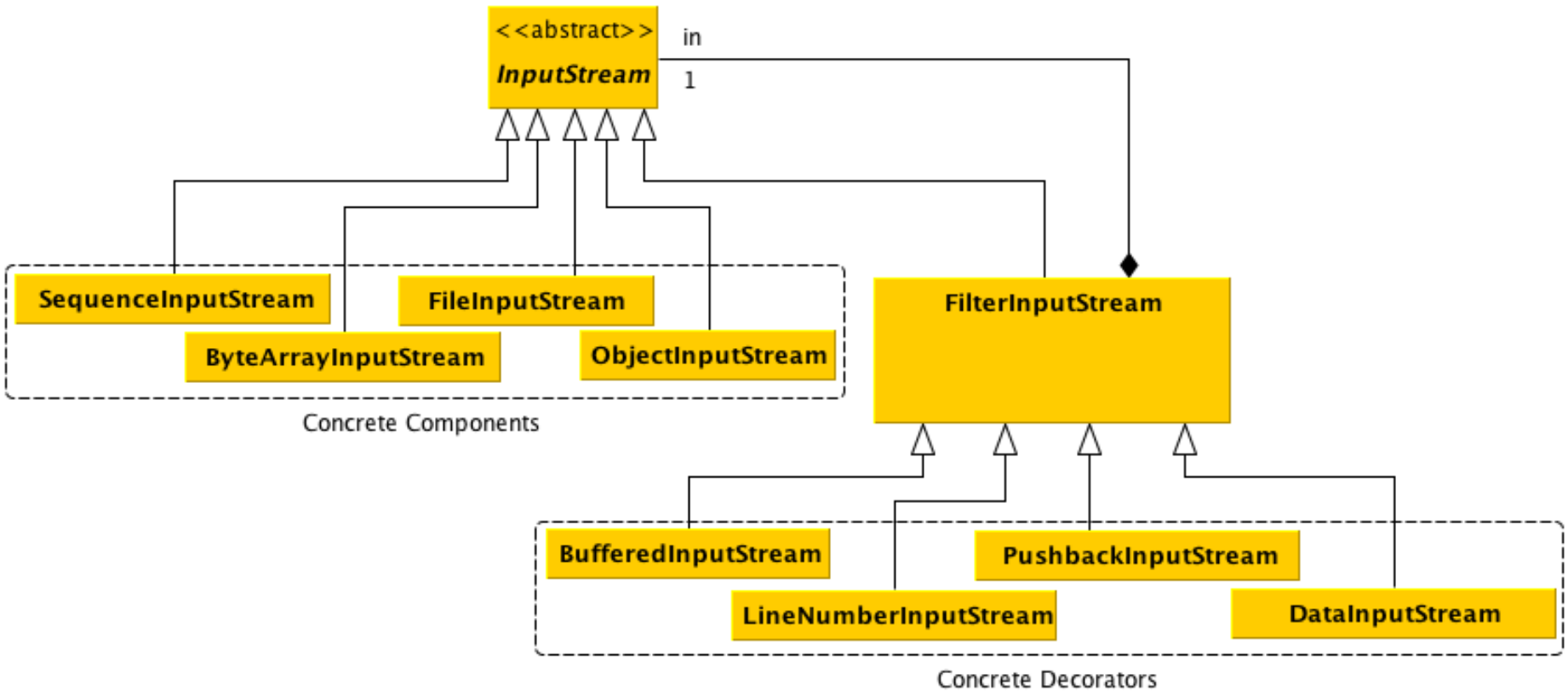
IO Streams in Java

Achtung

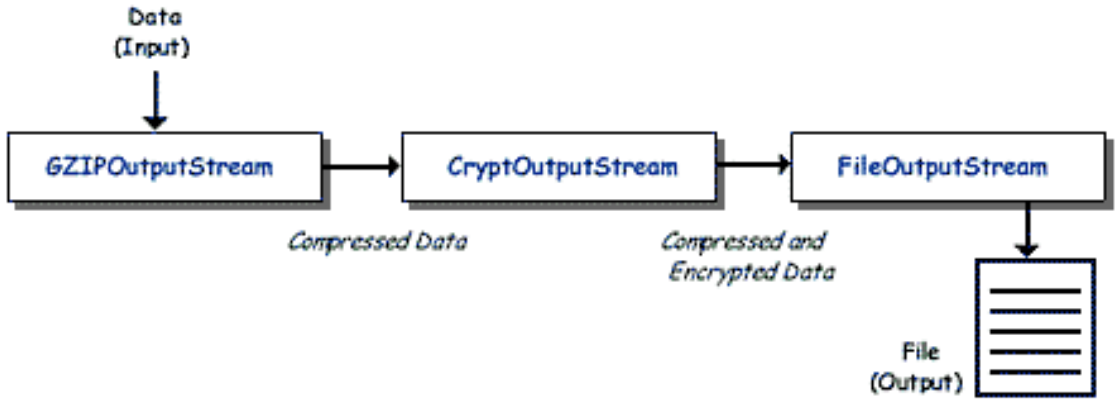
IO Streams sind nicht mit der Stream API (wird in einer späteren Vorlesung behandelt) ab Java 8 zu verwechseln!

- InputStream - Abstrakte Klasse zum Lesen von Byteströmen aus einem imaginären Eingabegerät
- OutputStream - Abstrakte Klasse zum Schreiben von Byteströmen in ein imaginäres Ausgabegerät
- Konkrete Unterklassen realisieren die Zugriffe auf echte Ein- und Ausgabegeräte, z.B. Dateien, Strings oder Netzwerkkanäle
- Stream-Klassen können verkettet werden, um Filter oder andere Zusatzfunktionen in den Streaming-Vorgang zu integrieren, z.B. Puffern von Zeichen, Zählen von Zeilen, Komprimierung o.ä. (“stream chaining”)

Typhierarchie für InputStream



Stream Chaining




```
FileOutputStream fos = new FileOutputStream("f.out");
CryptOutputStream cos = new CryptOutputStream(fos);
GZIPOutputStream gos = new GZIPOutputStream(cos);
gos.write('a');
gos.close();
```

- Verkettung durch Übergabe in den Konstruktor des nächsten Streams
- Methodenaufrufe kaskadieren durch die Kette, d.h. es wird sequentiell gezippt, verschlüsselt und dann in die Datei geschrieben

java.io.Writer

- Die abstrakte Klasse Writer ist die Basisklasse aller Character-Stream-Ausgaben des JDK
- Das JDK liefert konkrete Implementierungen mit, z.B. OutputStreamWriter, FileWriter, BufferedWriter, StringWriter etc.

Zentrale Methoden von Writer

```
// Flushen und danach den Writer schliessen
abstract public void close()

// Leert eventuell vorhandene Puffer
abstract public void flush()

// Diverse write-Methoden
public void write(int c)
public void write(char cbuf[])
abstract public void write(char cbuf[], int off, int len)
public void write(String str)
public void write(String str, int off, int len)
```

Beispiel für einfaches Schreiben einer Datei

```
FileWriter f;
String s = "Hallo␣Welt!";

try {
    f = new FileWriter("hallo.txt");
    f.write(s);

} catch (IOException e) {
    System.out.println("Fehler␣beim␣Schreiben!");

} finally {
    if (f != null)
        try {
            f.close();
        } catch (IOException e) {
            System.out.println("Fehler␣beim␣Schliessen");
        }
}
```

java.io.Writer - Schreiben mit Resource-try

try with resources

```
String s = "Hallo␣Welt!";

try(FileWriter f = new FileWriter("hallo.txt")){
    f.write(s);
} catch (IOException e) {
    System.out.println("Fehler␣beim␣Schreiben!");
}
```

- Neu in Java 1.7 ist try with resources.
- Vereinfacht das close von Resourcen.
- Eine Resource muss Closeable implementieren.
- IO Dateien, SQL Statements, eigene Klassen die Closeable implementieren etc.

Datentyp List in eine Datei schreiben

```
public void writeList(List<String> list) {

    try (PrintWriter out =
            new PrintWriter(new FileWriter("OutFile.txt"))){

        for (int i = 0; i < list.size(); i++)
            out.println("Value␣" + i + "=" + list.get(i));

    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("ArrayIndexOutOfBounds");

    } catch (IOException e) {
        System.err.println("IOException");
    }
}
```

OS Abhänigkeiten

- Dateinamen sind bei manchen Betriebssystemen case sensitiv (Linux, Mac OS), bei anderen nicht (Windows).
- Zum Navigieren in Verzeichnissen File.separator (ist \ oder /) verwenden.
- Zum Konstruieren von Path Angaben mit mehreren Verzeichnissen File.pathSeparator (ist ; oder :)verwenden.

java.io.Reader

- Die abstrakte Klasse Reader ist die Basisklasse aller Character-Stream-Eingaben des JDK
- Das JDK liefert konkrete Implementierungen mit, z.B. FileReader, InputStreamReader, BufferedReader, CharArrayReader etc.

Zentrale Methoden von Reader

```
// Den Reader schliessen
abstract public void close()

// Diverse read-Methoden
public int read()
public int read(char cbuf[])
abstract public int read(char cbuf[], int off, int len)
```

Beispiel für gepuffertes Einlesen einer Datei

```
BufferedReader r;
String l;
try {
    r = new BufferedReader(new FileReader("setup.log"));
    while ((l = r.readLine()) != null) {
        System.out.println(l);
    }

} catch (IOException e) {
    System.out.println("Fehler beim Lesen!");

} finally {
    if (r != null)
        try {
            r.close();
        } catch (IOException e) {
            System.out.println("Fehler beim Schliessen!");
        }
}
```

java.io.Reader - Lesen mit Resource-try

Auch beim Lesen einer Datei sollte ab Java 1.7 resource-try verwendet werden:

Lesen mit resource-try

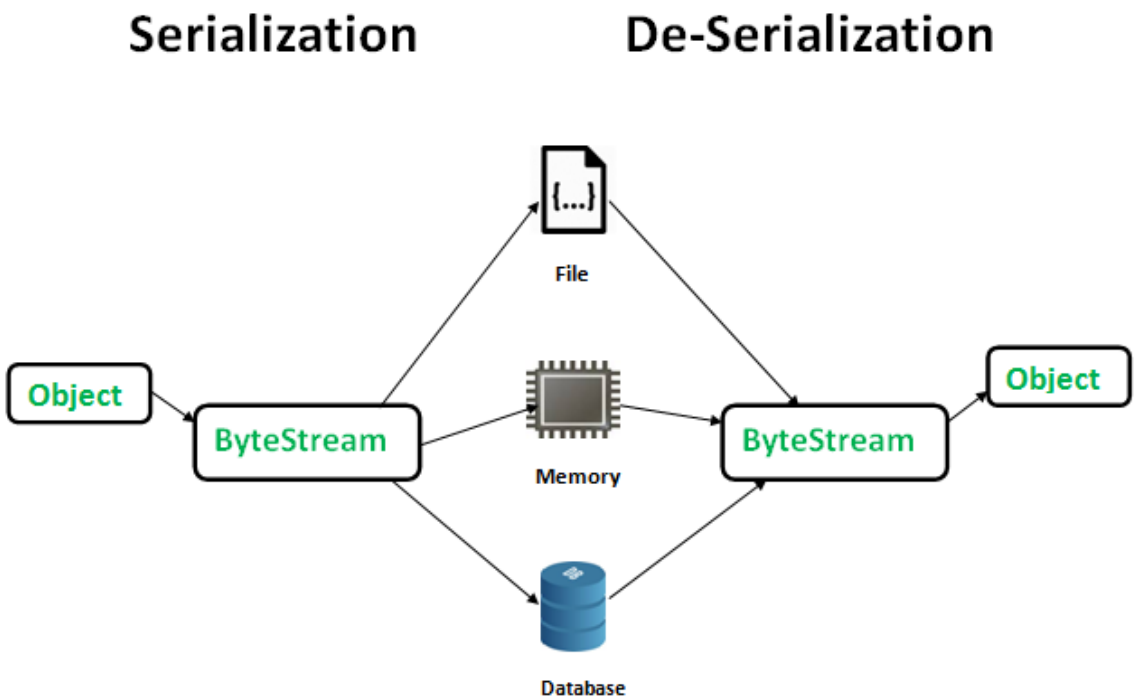
```
String l;
try(BufferedReader r =
        new BufferedReader(new FileReader("setup.log"))) {

    while ((l = r.readLine()) != null) {
        System.out.println(l);
    }

} catch (IOException e) {
    System.out.println("Fehler␣beim␣Lesen!");
}
```

Schreiben/Lesen von Objekten in Dateien

- Möchte man Objekte in eine Datei speichern, muss man diese in eine Sequenz von Bytes überführen.
- Dieser Vorgang muss systematisch erfolgen, damit Java beim Lesen der Datei den Zustand des Objektes rekonstruieren kann.
- Java hat einen Standardmechanismus für Entwickler mit der Serialization-API.



Serialization

- Objektserialisierung ist der Prozess zum Speichern des Zustands eines Objektes in eine Sequenz von Bytes.
- Der Prozess ermöglicht auch das Wiederherstellen des Objektes mit seinem Zustand aus der Bytesequenz.
- Serialisierbare Objekte werden durch das Marker Interface Serializable beschrieben.
- Serialisierung erzeugt eine tiefe Kopie $\Rightarrow$ alle Objekte dieser Kopie müssen serialisierbar sein.
- Java nutzt intern die Methoden `writeObject(ObjectOutputStream oos)` und `readObject(ObjectInputStream ois)`. Durch die Implementierung dieser Methoden im zu serialisierenden Objekt, kann das Verhalten gesteuert werden.

Object Streams

- Object Streams können benutzt werden, um serialisierbare Objekte zu speichern oder zu laden.

Beispiel (unvollständig)

```
public void writeMyObject(Serializable aSerializable)
            throws Exception {
  fos = new FileOutputStream("file.tmp");
  oos = new ObjectOutputStream(fos);
  oos.writeObject(aSerializable);
}

public Object readMyObject() throws Exception {
  fis = new FileInputStream("file.tmp");
  ois = new ObjectInputStream(fis);
  return ois.readObject();
}
```

Variable Argumentenanzahl

Motivation

- Will man einer Methode mehrere Objekte eines Typs übergeben, so packt man die Objekte in ein Array des Typs ein und übergibt das Array als ein Parameter.
- Das Varargs Feature des Compilers ermöglicht die Automatisierung der Arrayerstellung, so dass der Programmierer nichts mehr von der Array Erstellung merkt.

Varargs Beispiel

```
//Klassischer Ansatz
public void calculate(int[] i){ for (int a:i) do(a);}
int[] arr = new int[]{2,3,4}
calculate(arr);

// Varargs Feature
public void calculate(int... i){ for (int a:i) do(a);}
calculate(2,3,4);
```

Generics

Wir brauchen generische Typen!

Java ist statisch getypt, damit keine “message not understood” Fehler auftreten.

Genau das passiert aber bei der Arbeit mit Collections

```
List productList = new ArrayList();

// Products der Liste hinzufügen
productList.add(new Material());
productList.add(new Service());

// Autsch, welcher Idiot passt denn da nicht auf
productList.add(new String("Material"));

// Geben wir mal die ProductList aus
for(Object o:productList){
   ((IProduct)o).ausgeben(); // beim 3. knallts
}
```

Hurra, seit Java 5 gibt es Generics!

Java ohne Generics

```
public class Collections{
...
static Object max(Collection coll){...}
...
}
```

Java mit Generics

```
static <T extends Object & Comparable<? super T>> T
       max(Collection<? extends T> coll)
```

Java Generics

- Seit Java 5 ist es möglich Collections zu parametrisieren und damit zur Entwicklungszeit festzulegen, dass nur Objekte eines gewünschten Typs aufgenommen werden können.
- Der Compiler (und das Laufzeitsystem!) stellen sicher, dass nur Objekte eines erlaubten Typs in eine Collection aufgenommen werden.
- Typcasts beim Auslesen (und das damit verbundene Risiko von Laufzeitfehlern) entfallen
- Programmcode wird lesbarer, kürzer und fehlerfreier

Einfaches Beispiel

```
List<Integer> list = new ArrayList<Integer>();

int sum;
for (Integer i : list) sum+=i;
```

Klassendeklaration und Sprechweisen

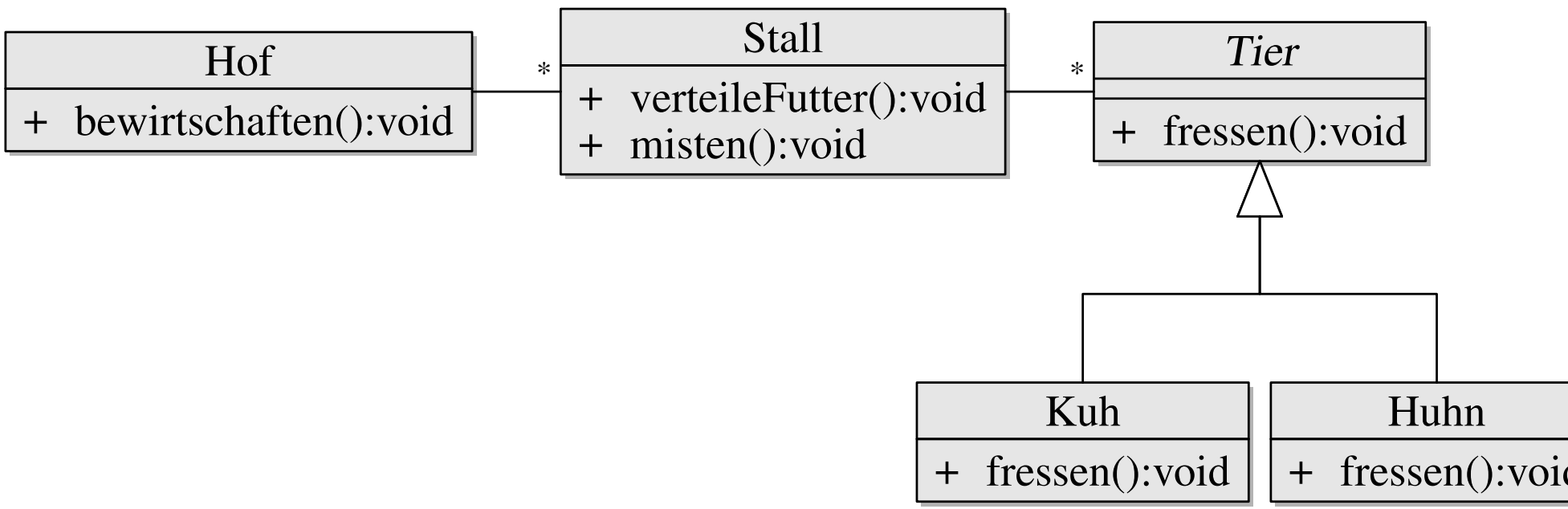
Deklaration

```
public interface List<E> {
 ...
 public void add(E element){...};
 public E get(int i){...};
 ...
}
```

Begriff	**englisch**	**Beispiel**
Generischer Typ	generic type	List<E>
Typvariable oder formaler Typparameter	formal type parameter	E
Parametrisierter Typ	parametrized type	List<Integer>
Typparameter	actual type parameter	Integer
Original Typ	raw type	List

Das Beispiel Tierleben

- Es können beliebige Klassen und Interfaces parametrisiert werden, welche dann nur mit Objekten eines bestimmten Typs zusammenarbeiten.
- Ställe sollen z.B. nur eine Tierart beherbergen.



Das Beispiel Tierleben

Ohne Generics

```
public class Stall {
  private List tiere;

  public Stall(Tier... t) {
    tiere = Arrays.asList(t);
  }
  public void verteileFutter() {
    for (Object o: tiere) ((Tier)o).fressen();
  }
}
```

Multi Kulti Ställe

```
    Stall s = new Stall(new Kuh("Clara"),
                        new Huhn("Picki"));
```

Monokultur Ställe

Es ist möglich den Typparameter mittels extends auf einen engeren Klassenbaum einzuschränken

Mit Generics

```
public class Stall<T extends Tier> {
  private List<T> tiere;
  public Stall(T... tiere) {
    this.tiere = Arrays.asList(tiere);
  }
  public void verteileFutter() {
    for (T tier: tiere) tier.fressen();
}}
```

Monokultur-Ställe

```
Stall<Kuh> s = new Stall<Kuh>(new Kuh("Clara"),
                              new Kuh("Bella"));
```

Diamond Operator

Diamonds are forever...

```
Stall<Kuh> s = new Stall<>();

public Stall<Kuh> baueStall(){
    return new Stall<>();
}
```

- Redundante Typangaben können weggelassen werden. Das macht den Code lesbarer.
- Sprachkonstrukt seit Java 7.
- Der Operator kann immer dann eingesetzt werden, wenn durch den umgebenden Context klar ist, mit welchem Typ ein Generic parametrisiert werden soll.

Mehrere Typparameter 1/2

- Es ist möglich eine Klasse mit mehreren Typparametern auszustatten, z.B. bei Map<K, V> wobei K die Typvariable des Key-Objekts und V die des Value-Objekts ist.
- Typparameter können selbst parametrisierte Typen sein.

Beispiel

```
Map<Integer, Stall<Kuh>> m = new HashMap<>();

Stall<Kuh> s1 = new Stall<>(new Kuh("Clara"),
                            new Kuh("Bella"));

Stall<Kuh> s2 = new Stall<>(new Kuh("Polly"));

Stall<Pferd> s3 = new Stall<>(new Pferd("Jolly␣Jumper"));

Stall s4 = new Stall(); // Raw type warning
```

Mehrere Typparameter 2/2

Beispiel

```
...
m.put(1, s1); // Ok
m.put(2, s2); // Ok
m.put(3, s3); // Compilefehler
m.put(4, s4); // Unchecked conversion warning
```

Zwei wesentliche Fehler bei der Verwendung von Generics:

1. Versucht man ein Objekt vom falschen Typ in eine parametrisierte Collection oder ein parametrisiertes Objekt "hineinzustecken" reagiert der Compiler unmittelbar mit **einem Fehler**.
2. Verwendet man an einer Stelle, wo ein parametrisierter Typ erwartet wird, einen Originaltyp (="raw"), erzeugt der Compiler **eine Warnung**, dass eine Konvertierung zur Laufzeit fehlschlagen kann. Der Code kann jedoch trotzdem kompiliert werden.

Namenskonventionen für Typvariablen

Folgende Namenskonventionen für Typvariablen gelten

- Typvariablen werden mit einem einzelnen Großbuchstaben bezeichnet
- K,V = Key und Value (bei Schlüssel-Wert-Paaren)
- E = Element (bei Collections)
- T = Typ (wenn nichts von obigem zutrifft)

Klassendeklaration

```
public class MiniMap<K,V> {...}
public class EmptyList<E> {...}
public class Stall<T extends Tier> {...}
```

Fehlende Kovarianz

- Parametrisierte Typen sind nicht in einer Vererbungshierarchie, auch wenn Ihre Typparameter in einer Vererbungshierarchie stehen!
- Wäre dies der Fall, würde die Typsicherheit verletzt werden.

Das haben Sie nicht erwartet, oder?

```
Stall<Kuh> ks = new Stall<>();
Stall<Tier> ts = ks; // Compilefehler, sonst...

ts.add(new Meerschweinchen());
Kuh clara = ks.get(0); // Typverletzung!
```

Wildcards

- Problem: Ein Stall kann mit jeder Art Tier ausgemistet werden. In der unten gezeigten Methode würde der Aufruf mit einem Objekt vom Typ `Stall<Kuh>` jedoch zum Compile-Error führen.
- Es fehlt ein gemeinsamer Obertyp für die parametrisierten Klassen.
- Dieses Beispiel ist nicht generisch genug.

Klappt nur mit Tierställen

```
public class Hof {
    public void miste(Stall<Tier> ts) {
        ts.misten();
    }
}
```

Wildcards

- Die Wildcard “?” ermöglicht es, eine **gemeinsame** Oberklasse für alle parametrisierten Klassen zu definieren.
- Wird mit einer Wildcard parametrisiert, kann man mit new keine Exemplare dieser Klasse erzeugen, ähnlich wie bei abstrakten Klassen.
- Verwendet man Wildcards beschränkt der Compiler allerdings die Collection/Klasse auf lesende Operationen, um Typverletzungen zur Laufzeit zu verhindern.

Klappt mit allen Ställen

```
public void miste(Stall<?> s) {
  s.misten();
  s.neuesTier(new Kuh("Polly")); // <--
        // geht nicht, da nicht lesend
}
```

Die Spitze des Eisbergs

Was gibt es darüber hinaus?

- Vollständige Regeln, wie generische Typen definiert und parametrisiert werden.
- Type Erasure
- Sonderfälle
- Kompatibilität mit Legacy Java Anwendungen
- Laufzeitaspekte (Neuerungen in der VM)
- Typinferenz
- super, ...
- u.v.a.m.

Aufgabe

Übung IO Streams und Generics 1/3

In dieser Aufgabe kommen wir noch einmal auf die Bestelllistenapplikation zurück und wollen sie noch ein wenig refactoren und erweitern. Für die folgenden Aufgaben benötigen Sie keine Libraries.

1. Nutzen Sie entweder als Basis Ihre Lösung von letzter Woche oder importieren Sie die Musterlösung im gitlab als neues Java-Projekt.
2. Ersetzen Sie die setter Methode für die factories Variable in der Bestellliste durch eine Methode, welche VarArgs benutzt und vereinfachen Sie die main Methode.
3. Nutzen Sie in der Übung statt Arrays konsequent parametrisierte Collections (wie z.B. Listen). Passen Sie dazu auch alle bereits bestehenden Arrays an.

Übung IO Streams und Generics 2/3

4. Erweitern Sie das Menü um einen Punkt “Speichern”. Ermöglichen Sie damit das Speichern einer Musterbestellung in einer Datei. Beim Speichern soll ein Dateiname vom Nutzer abgefragt werden.
5. Erweitern Sie das Menü um einen Punkt “Laden”. Ermöglichen Sie das Laden einer Musterbestellung aus einer Datei. Dazu wird zunächst nach einem Dateinamen gefragt.
6. Stellen Sie durch ordentliches Exception-Handling sicher, dass keine Programmabbrüche auftreten können, wenn z.B. die Datei beim Ladeversuch nicht existiert oder beim Speichern die Festplatte voll ist.

Übung IO Streams und Generics 3/3

Zusatzaufgabe

- Schreiben Sie ein Java-Programm, dass eine UTF-8 codierte Datei einliest und als eine ISO-8859-1 codierten Datei ausgibt.

Zusatzaufgabe - Achtung sehr schwer!

Importieren Sie Gson (per Maven) und ändern das Format der gespeicherten Datei in JSON. Implementieren Sie das Einlesen und die Speicherung in einer eigenen Klasse.

- Implementieren Sie die Speicherung in JSON in der Datei.
- Implementieren Sie das Einlesen von JSON aus der Datei! Tun Sie dafür, was nötig ist...

Achtung! Polymorphie!