4. Parametrisierbare Klassen

Java now supports *generics*, the most significant change to the language since the addition of inner classes in Java 1.2 — some would say the most significant change to the language ever.

M. Naftalin, P. Wadler, *Java Generics*, O'Reilly, 2006.

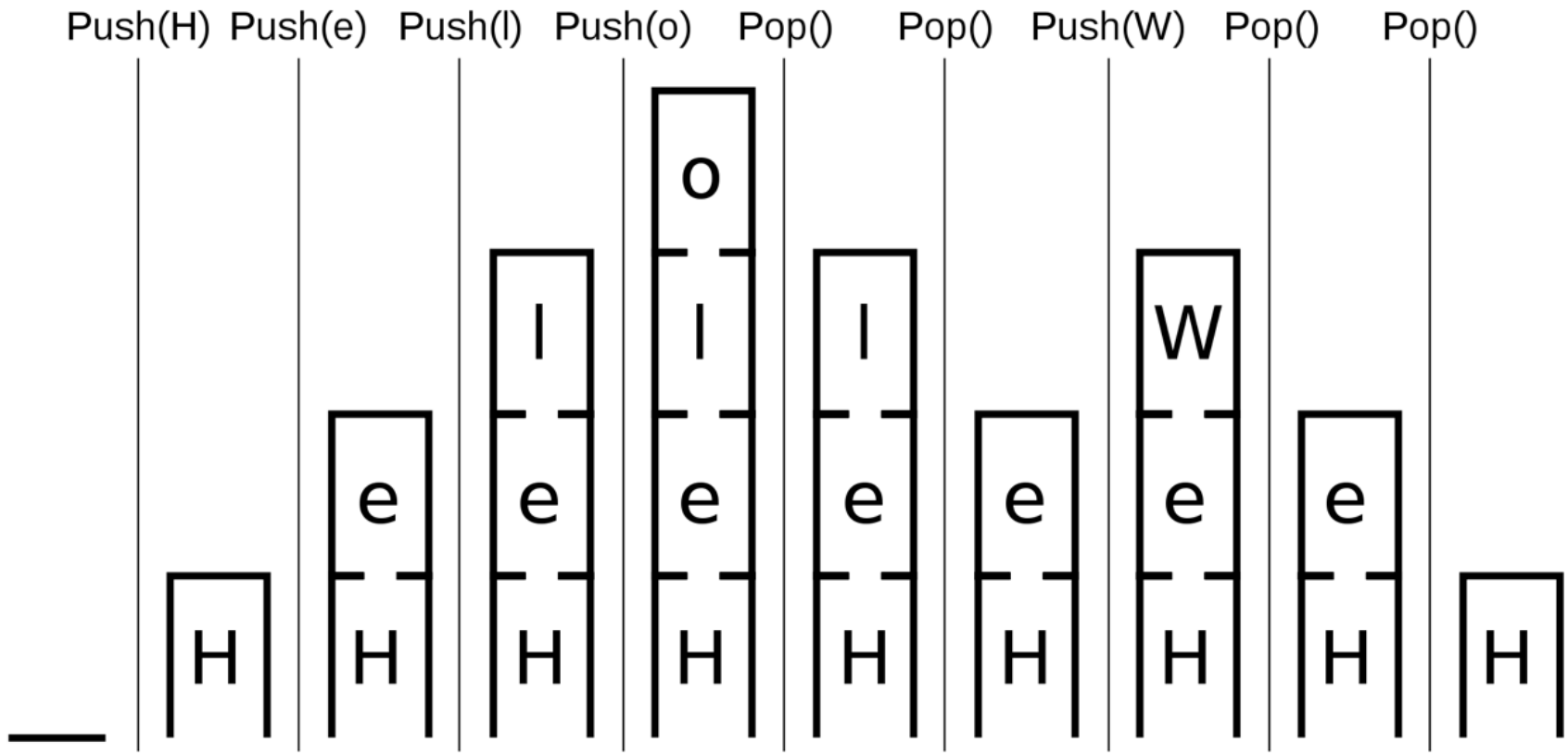
Datenstruktur Stapel (Stack)

Wichtige abstrakte Datentypen bzw. Datenstrukturen der Informatik sind unabhängig von einem Basis- bzw. Komponententyp.

Definition 4.1. Es sei T ein beliebiger Datentyp. Ein *Stapel (Stack)* ist eine Datenstruktur, die folgende Operationen unterstützt:

- `void push(T elem)`
  Legt ein Element $elem$ vom Typ T auf dem Stapel ab.
- `T pop()`
  Entfernt das oberste Element vom Stapel und liefert es als Ergebnis zurück.
- `T peek()`
  Liefert das oberste Element des Stapels, ohne es vom Stapel zu entfernen.

Stapel (Stack)



Naiver Implementierungsansatz

- Für jeden Komponententyp wird der Stapel separat implementiert.
- Die Anweisungen zur Implementierung eines Stapels sind nahezu unabhängig vom Komponententyp T.
- Vergleicht man die Implementierungen eines Stapels z.B. für `int` und `String`, so unterscheiden sich die Implementierungen nur im angegebenen Datentyp.

☞ Beispiel

Ursprünglicher generischer Implementierungsansatz: `Object`

Allgemeine Objektreferenz mittels `Object`.

```
public class Stapel
{
   ...
   void push(Object element) { ... }
   Object pop() { ... }
   Object peek() { ... }
   ...
}
```

☞ Vorgehensweise bis Java 1.4

Generische Objektreferenz mittels `Object` (1)

Probleme:

- Downcasting notwendig nach `pop()` und `peek()`

```
T t = (T) stapel.pop();
```

- Keine Typüberprüfung zur Übersetzungszeit

```
Stapel stapel = new Stapel();
stapel.push(new Integer(4711)); // wird vom Compiler trotz
stapel.push("hallo!");          // verschiedener Typen akzeptiert
```

- Evtl. Typfehler zur Laufzeit

```
stapel.push("hallo!");
Integer i = (Integer) stapel.pop(); // ClassCastException
```

Generische Objektreferenz mittels `Object` (2)

Und wenn wir doch für jeden benötigten Komponententyp eine eigene Implementierung bereitstellen?

- hoher Aufwand
  - Mehrfache Verwendung des fast gleichen Quelltextes
  - Mehrfaches Testen notwendig
  - entdeckte Fehler müssen mehrfach korrigiert werden
- geringe Abstraktion

Typvariablen

- Seit Java 5 sind *Typvariablen* bei einer Klassendefinition möglich.
- Die damit realisierte Klasse entspricht einem *generischen Datentyp*.
- In der objektorientierten Programmierung bezeichnen wir dies auch als *parametrische Polymorphie*.
- In Java bezeichnet man diese Möglichkeit als *Generics*.

Wirkung:

- Typvariablen ermöglichen es, bei der Definition komplexer Datentypen von den zu Grunde liegenden Basistypen zu abstrahieren.
- Erst bei der Instanziierung muss für die Typvariablen ein konkreter Typ angegeben werden.

Typvariablen: Syntax

```
public class Stapel<T>
{
    ...
    void push(T element) { ... }
    T pop() { ... }
    T peek() { ... }
    ...
}
```

- Hier eine Typvariable T, angegeben in spitzen Klammern hinter dem Klassennamen
- Die Typvariable T steht innerhalb der Klassendefinition für einen beliebigen Referenztyp.
- Innerhalb des Quelltextes einer generischen Klasse kann die Typvariable (fast) überall dort stehen, wo ein Datentyp verlangt ist.

Instanziierung generischer Typen (1)

Bei der Instanziierung eines generischen Typs müssen wir die Typvariable durch einen konkreten Typ ersetzen.

```
Stapel<String> sstapel = new Stapel<String>();
Stapel<Double> dstapel = new Stapel<Double>();
```

Die Stapel sind jetzt typsicher:

```
dstapel.push("text");   // hier meckert der Compiler
```

Kein Downcast notwendig:

```
dstapel.push(new Double(Math.PI));
Double d = dstapel.peek();          // auch ohne Downcast kein Problem
```

Instanziierung generischer Typen (2)

Leider wäre die folgende Deklaration nicht erlaubt:

```
Stapel<int> istapel = new Stapel<int>();
```

Grund: Typvariablen dürfen nur durch Referenztypen instanziiert werden.

Wie können wir aber dann einen Integer-Stapel erzeugen? Verwendung von Wrapper-Klassen: `Integer`, `Double`, etc.

```
Stapel<Integer> istapel = new Stapel<Integer>();
```

Exkurs: Wrapper-Klassen (Hüllklassen)

- Instanzen von *Wrapper-Klassen (Hüllklassen)* haben die Aufgabe, einen primitiven Wert als Objekt zu repräsentieren.
- Es gibt für jeden einfachen Datentyp eine zugehörige Wrapper-Klasse, z.B. `int/Integer`, `double/Double`, `char/Character`.
- `Integer i = new Integer(4711);`
- Wie Strings sind Instanzen von Wrapper-Klassen grundsätzlich *unveränderlich (immutable)*.

Exkurs: Boxing und Unboxing

- Die Repräsentation eines einfachen Wertes als Objekt mit Hilfe einer Wrapper-Klasse bezeichnen wir auch als *Boxing*.

```
Integer io = new Integer(4711); // Boxing
```

- Der Zugriff auf den einfachen Wert nennen wir *Unboxing*.

```
int ival = io.intValue(); // Unboxing
```

Exkurs: Autoboxing (1)

- Die manuelle Ausführung von Boxing und Unboxing ist oft unhandlich.

```
Stapel<Integer> istapel = new Stapel<Integer>();
istapel.push( new Integer(4711) );
int iv = istapel.peek().intValue();
```

- Die automatische Umwandlung von Werten einfacher Datentypen in Instanzen einer Wrapper-Klasse und umgekehrt wird als *Autoboxing* bezeichnet.
- Java beherrscht Autoboxing seit Java 5.

```
int i     = 4711;
Integer j = i;          // automatisches Boxing
int k     = j;          // automatisches Unboxing
```

Exkurs: Autoboxing (2)

Damit ist natürlich auch möglich:

```
Stapel<Integer> istapel = new Stapel<Integer>();
istapel.push(4711);          // Boxing
int iv = istapel.pop();      // Unboxing
```

Vorsicht bei Vergleichen mit == und Autoboxing!

Typanpassungen

Ein instanziierter generischer Typ lässt sich durch Typanpassung auf eine allgemeine Form bringen:

```
Stapel<Integer> istapel = new Stapel<Integer>();
Stapel          stapel  = (Stapel) istapel;
```

Jetzt findet für `stapel` keine Typprüfung mehr statt. Daher würde

```
stapel.push("No Integer");
```

keinen Fehler zur Übersetzungszeit liefern.

Typvariablen in Methoden

Typvariablen können auch auf Methodendeklarationen beschränkt sein:

```
public class Util
{
   public static <T> T zufall(T o1, T o2)
   {
      return Math.random() < 0.5 ? o1 : o2;
   }
}
```

Die Angabe von <T> beim Klassennamen entfällt und verschiebt sich auf die Methodendefinition.

☞ Typinferenz (Schlussfolgerungen über Datentypen), siehe Beispiel

Mehrere Typvariablen

- Bei der Definition einer generischen Klasse können auch mehrere Typvariablen verwendet werden.
- Beispiel: Eine generische Klasse für die Repräsentation einer Funktion

$$f : T \longrightarrow U$$

  mit endlichem Definitionsbereich T und Wertebereich U.
- entspricht einer funktionalen endlichen Relation $T \times U$.

```
public class Zuordnung <T,U> {
    ...
    void put(T t, U u) { ... }
    U get(T t) { ... }
    ...
}
```

- put: ordnet einem Wert $t \in T$ den Funktionswert $u \in U$ zu
- get: liefert zu einem Wert $t \in T$ den zugehörigen Funktionswert $f(t) \in U$
- Beispielhafte Nutzung:

```
Zuordnung<Integer,String> z = new Zuordnung<Integer,String>();
                          // Definition der Zuordnungen:
z.put(1,"eins");          // 1 --> "eins"
z.put(2,"zwei");          // 2 --> "zwei"

String s2 = z.get(2);     // liefert "zwei"
```

Schachtelung von Generischen Typen

- Bei der Instanziierung von generischen Klassen kann als Typ auch eine instanziierte generische Klasse verwendet werden.
- Beispiel: Ein Stapel von String-Listen:

```
Stapel<ArrayList<String>> stapelVonListen
                              = new Stapel<ArrayList<String>>();
```

Die new-Operation erzeugt natürlich wiederum nur den Stapel, aber nicht die im Stapel ablegbaren Listen. Diese müssen separat erzeugt werden.

```
// neue Liste erzeugen und auf Stapel ablegen
stapelVonListen.push(new ArrayList<String>());

// Liefert oberste Liste des Stapels
ArrayList<String> list = stapelVonListen.peek();
```

Typvariablen und Schnittstellen

- Typvariablen dürfen auch bei der Definition von Schnittstellen verwendet werden. Damit enststehen generische Schnittstellen.
- Syntax für Typvariable analog zu Klassen:

```
public interface List<T> {
   void add(T elem);  // Element hinzufuegen
   int size();        // Anzahl Listenelemente bestimmen
   T get(int i);      // Liefert das i-te Element
   ...
}
```

Die Schnittstelle Comparable

- `Comparable<T>` (aus dem Paket `java.lang`) ist eine generische Schnittstelle für die Vergleichbarkeit von Objekten eines Typs T.
- Definition mit Generics (vgl. Folie 62):

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

- Es sei `this` das Objekt, für das die Methode `compareTo()` aufgerufen wird. Dann soll `compareTo()` liefern:
  - $== 0$, wenn `this` und `o` gleich sind,
  - > 0, wenn `this` größer als `o` ist und
  - < 0, wenn `this` kleiner als `o` ist.

- Einige wichtige Klassen des Java-API implementieren `Comparable<T>`, z.B. die Wrapper-Klassen.
- D.h. die Klasse `String` implementiert `Comparable<String>`, `Integer` implementiert `Comparable<Integer>`, usw.
- Vgl. Übungsaufgabe 1 (a) auf Aufgabenblatt 3: Mit Verwendung generischer Typen

```
public class Person implements Comparable<Person> {
   ...
   public int compareTo(Person o) {
      ...
   }
   ...
}
```

- Mit Hilfe von `Comparable<T>` können generische Sortierverfahren implementiert werden.
  ☞ Übungsaufgabe
- Eine ähnliche Schnittstelle ist `Comparator<T>` aus dem Paket `java.util`.

Die Schnittstelle Comparator

Problem der Schnittstelle `Comparable`:

- Sie muss von der Klasse implementiert werden, deren Instanzen man vergleichen möchte.
- Damit ist in der Klasse nur ein Sortierkriterium definierbar.
- Beispiel: Übungsaufgabe 1, Blatt 3: `Person`-Instanzen können nur nach Namen aber nicht z.B. nach PLZ sortiert werden.

Lösung:

- Klassenlogik und Vergleichslogik trennen.
- Komparatoren definieren, die einen Vergleich von Instanzen einer Klasse K nach eigener Logik durchführen.
- Jeder Komparator kann die Instanzen von K auf andere Weise vergleichen.

Die Schnittstelle Comparator (2)

Generische Schnittstelle `Comparator<T>` aus dem Paket `java.util`:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    boolean equals(Object o);
}
```

`compare(T o1, T o2)` folgt der üblichen Logik und liefert:

- $== 0$, wenn o1 gleich o2 ist,
- > 0, wenn o1 größer als o2 ist und
- < 0, wenn o1 kleiner als o2 ist.

Die Schnittstelle Comparator (3)

Man beachte:

- Die Methode `compare()` wird nicht auf einer Instanz der Klasse `T` aufgerufen, sondern auf einer Komparator-Instanz, die `Comparator<T>` implementiert.
- Wir können nun ganz unterschiedliche Klassen für unterschiedliche Arten des Vergleich implementieren.

Beispiel:

- Für einen Vergleich von `Person`-Instanzen nach Namen:

```
public class NameKomparator implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        ...
    }
    ...
}
```

- Für einen Vergleich von `Person`-Instanzen nach Postleitzahlen:

```
public class PLZKomparator implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        ...
    }
    ...
}
```

- Das Java SDK bietet uns generische Sortiermethoden, wobei die Sortierung über einen Komparator gesteuert werden kann (z.B. in der Klasse `java.util.Arrays`, ungefähr so):

```
public class Arrays {
    ...
    public static <T> void sort(T[] a, Comparator<T> c) { ... }
    ...
}
```

- Dies können wir nun wie folgt nutzen:

```
public class Test {
    public static void main(String[] args) {
        Person[] p = new Person[...];
        p[0] = ... // Feld fuellen;
        ...
        Comparator<Person> c1 = new NameKomparator();
        Comparator<Person> c2 = new PLZKomparator();

        Arrays.sort(p,c1);   // sortiert Feld p nach Namen
        Arrays.sort(p,c2);   // sortiert Feld p nach Postleitzahlen
    }
}
```

Die Schnittstellen Iterable und Iterator

- java.lang.Iterable:

```
public interface Iterable<E> {
    public Iterator<E> iterator();
}
```

- java.lang.Iterator:

```
public interface Iterator<E> {
    public boolean hasNext();
    public E next();
    public void remove();
}
```

- Diese Schnittstellen dienen dazu, auf die Elemente einer Kollektion iterativ zuzugreifen.
- Alle Kollektionen des Java Collection Frameworks implementieren `Iterable`, z.B. `ArrayList`.
- Auch die Schnittstelle `List` ist von `Iterable` abgeleitet.

Nutzung von Iterator

```
List<String> list = new ArrayList<String>();
list.add("eins"); list.add("zwei"); list.add("drei");

Iterator<String> iter = list.iterator();
while ( iter.hasNext() ) {            // Elemente der Liste sequentiell ausgeben
    System.out.println(iter.next());
}
```

Foreach

Java bietet für den iterativen lesenden Zugriff auf eine Kollektion eine spezielle Form der `for`-Schleife:

```
for (Datentyp Schleifenvariable : Kollektion )
    Anweisung
```

Voraussetzungen:

- *`Kollektion`* muss die Schnittstelle `Iterable` implementieren.
- *`Datentyp`* muss zum Basisdatentyp von *`Kollektion`* passen.

Wirkung:

- Es wird implizit ein Iterator erzeugt, mit dem die Kollektion vollständig durchlaufen wird.
- Jedes Element der Kollektion (vgl. `next()` von `Iterator`) wird dabei genau einmal an die Schleifenvariable gebunden.

Beispiele:

```
List<String> list = new ArrayList<String>();
list.add("eins"); list.add("zwei"); list.add("drei");

for (String s : list) {
    System.out.println(s);
}
```

Die Schleife wird dabei implizit ausgeführt als:

```
for (Iterator<String> iter=list.iterator() ; iter.hasNext() ; ) {
    String s = iter.next();
    System.out.println(s);
}
```

Auch für Felder kann diese Form der Schleife genutzt werden:

```
public static double sumArray(double[] a) {
    double s = 0.0;
    for (double x : a) {
        s += x;
    }
    return s;
}
```