Java Generics

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de, Raum 149B
Version: 1.0.1

Folien: https://delors.github.io/prog-adv-java-generics/folien.de.rst.html
https://delors.github.io/prog-adv-java-generics/folien.de.rst.html.pdf
Fehler melden: https://github.com/Delors/delors.github.io/issues

1. Einfache nicht-generische Datenstrukturen

Wiederholung von allg. Programmierkonzepten und Motivation für Generics.

Übung

1.1. Speichern von Wertepaaren

1. Implementieren Sie die Datenstruktur `Pair` die zwei Werte (vom Typ `Object`) speichern kann. Die Klasse soll folgende Methoden bereitstellen:

- `Pair(..., ...)`: Konstruktor zum Erzeugen eines `Pair`s.
- `getFirst()` und `getSecond()`: Liefert den ersten/zweiten Wert zurück
- `void setFirst(...)` und `void setSecond(...)`: Setzt den ersten/zweiten Wert
- `toString()`: Liefert eine String-Repräsentation des Paares

2. Erzeugen Sie ein `Pair`-Objekt mit zwei Integer-Werten (z. B. 1 und 2).
3. Nutzen Sie die Methoden der Klasse, um die Werte abzufragen und zu addieren.
4. Speichern Sie das Ergebnis an zweiter Stelle im `Pair`-Objekt als `String`.
5. Geben Sie den zweiten Wert auf der Konsole aus.
6. Was wäre bei der Addition passiert, wenn an zweiter Stelle ein `String` gespeichert gewesen wäre? Wann wäre das Verhalten aufgefallen?
7. Welchen statischen Typ hat der Wert, den die Methode `getFirst()` zurückgibt?

Übung

1.2. Datenstruktur zum Speichern von ganz vielen Werten

1. Implementieren Sie eine Datenstruktur `List` zum Speichern beliebig vieler Werte im Package `ds`. Die Klasse soll folgende Methoden bereitstellen:
   - `List()`: Konstruktor
   - `List(int size)`: Konstruktor
   - `void add(...)`: Fügt ein Element hinzu
   - eine *Varargs* Methode `void addAll(...)`:, die alle übergebenen Werte hinzufügt.
   - `int size()`: Liefert die Anzahl der Elemente zurück
   - `Object get(int index)`: Liefert das Element an der Stelle `index` zurück oder wirft eine `IndexOutOfBoundsException`, wenn der Index ungültig ist
   - `void set(int index, Object value)`: Setzt das Element an der Stelle `index` auf den Wert `value` oder wirft eine `IndexOutOfBoundsException`, wenn der Index ungültig ist
   - `String toString()`: Liefert eine String-Repräsentation der Liste
   - `void remove(int index)`: Entfernt das Element an der Stelle `index` oder wirft eine `IndexOutOfBoundsException`, wenn der Index ungültig ist
   - `void clear()`: Entfernt alle Elemente

   Nutzen Sie als zugrunde liegende Datenstruktur ein Array. D. h. speichern Sie die Elemente in einem Array und vergrößern Sie das Array, wenn es voll ist. Wenn das Array zu groß ist, verkleinern Sie es. Eine Vergrößerung soll das Array verdoppeln aber um nicht mehr als 1000 Elemente. Eine Verkleinerung soll das Array halbieren, wenn weniger als ein Viertel des Arrays belegt ist. Die Mindestgröße des Arrays soll 16 Elemente betragen.

   Nutzen Sie `java.lang.System.arraycopy(...)` zum Vergrößern/Verkleinern des zugrunde liegenden Arrays.
2. Schreiben Sie eine Klasse `ListTest`, die die Klasse `List` testet. Die Klasse soll jede der Methoden der Klasse `List` *testen*; d. h. mindestens einmal aufrufen.

Achtung!

Denken Sie an die Modellierung von Sichtbarkeiten.

Modellieren Sie alle Ausnahmen. Stellen Sie sicher, dass alle Methoden die Bedingungen einhalten.

Übung

1.3. `List` erweitern

1. Schreiben Sie eine Klasse `Stack` im Package `ds`, die Ihre Klasse `List` erweitert; d. h. von `List` erbt. Die Klasse soll folgende Methoden bereitstellen:
   - `Stack()`: Konstruktor
   - `void push(... value)`: Legt ein Element auf den Stack
   - `... pop()`: Entfernt das oberste Element vom Stack und liefert es zurück oder wirft eine `java.util.NoSuchElementException`, wenn der Stack leer ist
   - `... peek()`: Liefert das oberste Element zurück, ohne es zu entfernen oder wirft eine `NoSuchElementException`, wenn der Stack leer ist
2. Schreiben Sie eine Klasse RPN (im *Default Package*), die einen String von der Kommandozeile übernimmt (als einzelne "args") und diesen als *umgekehrte polnische Notation* interpretiert (d. h. erst kommen die Operanden, dann ein Operator) und berechnet.

   Beispielinteraktion:

   ```
   $ java --enable-preview RPN 1 2 "+" 3 "*"
   9.0
   ```

   Nutzen Sie Ihre Klasse `Stack` zum Zwischenspeichern der Operanden.
3. Wenn Sie sich den Code des RPN ansehen - wo sehen Sie insbesondere Verbesserungspotential?

Zusammenfassung

✓wir haben zwei grundlegende Datenstrukturen kennen gelernt sowie mögliche Implementierungen dafür:

1. Listen basierend auf Arrays
2. Stacks basierend auf Listen

! wir haben gesehen, dass die Verwendung von Datenstrukturen, die nichts über den Typ der gespeicherten Elemente wissen, zu Problemen führen kann (Fehler zur Laufzeit und nicht zur Compilezeit). Weiterhin sind viele explizite Typumwandlungen notwendig, die den Code unübersichtlich machen.

2. Generics - erste Einführung

Generics - Motivation

🔎 Beobachtung

- Eine Collection wie `Pair` oder `List` sollte zu mehr als einem Typ passen.
- Eine Implementierung sollte für verschiedene Zwecke ausreichend sein.
- Das allgemeine Verhalten der Collection hängt nicht vom Elementtyp ab.
- Zusätzlich wollen wir einen spezifischen Elementtyp garantiert haben.

  Angenommen, wir nutzen eine Collection nur für Person Instanzen, dann wollen wir auch Person Objekte verwenden können und nicht immer mit "Object" arbeiten müssen.

Generics erlauben die Definition generischer und typsicherer Datentypen, die über die Typen der Elemente abstrahieren.

D. h. wir können zum Beispiel angeben, dass wir nur Elemente vom statischen Typ `Person` speichern wollen. Dies hat folgende Vorteile:

✓Kompakterer / besser wartbarer Code

✓Fehler, die sonst erst zur Laufzeit auftreten würden, können zur Compilezeit erkannt werden (z. B. das versehentliche Speichern eines `String`s in einer Liste für `Integer` Werte.)

Generics - Beispiel: RPN Calculator mit verschiedenen `Stack`s

Verwendung eines einfachen `Stack`s ohne Typparametrisierung

```
1        Stack stack = new Stack();
2        for (String arg : args) {
3            switch (arg) {
4                case "+":
5                    stack.push((double) stack.pop() + (double) stack.pop());
6                    break;
7                case "*":
8                    stack.push((double) stack.pop() * (double) stack.pop());
9                    break;
10               default:
11                   stack.push(Double.parseDouble(arg));
12           }
13       }
```

In diesem Beispiel würden wir insbesondere gerne auf die Casts (2 Mal in Zeile 5 und 2 man in Zeile 8) verzichten wollen. Dies Casts sind nicht nur unschön, sondern können auch (in komplexeren Fällen) zu Laufzeitfehlern führen.

Verwendung eines `Stack`s für Double Werte

```
1        Stack<Double> stack = new Stack<>(); // ← Typ der Elemente ist Double
2        for (String arg : args) {
3            switch (arg) {
4                case "+":
5                    stack.push(stack.pop() + stack.pop());
6                    break;
7                case "*":
8                    stack.push(stack.pop() * stack.pop());
9                    break;
10               default:
11                   stack.push(Double.parseDouble(arg));
12           }
13       }
```

Einfache generische Klassen aus Java

```java
public interface Collection<E> {
    void add(E x);
    Iterator<E> iterator();
}

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

Mittels `<E>` oder `<T>` in der Klassendefinition deklarieren wird einen formalen Typparameter `E` bzw. `T`.

Dieser kann dann in der Klasse als Typ genutzt werden. Wenn wir dann eine Instanz der Klasse erzeugen, müssen wir den konkreten Typ für den Typparameter `E` bzw. `T` angeben.

Generics: Instanziierung

- Bei der Instanziierung von Generics muss für alle generischen Typen ein konkreter Datentyp (z.B. `Integer`) definiert werden:

  Beispiel

  ```
  List<Integer> v1 = new List<Integer>();
  ```

- Der konkrete Datentyp muss eine Klasse sein, d. h. es darf kein primitiver Datentyp (z.B. `int`) sein.
- Der konkrete Datentyp kann allerdings auch bei der Verwendung weggelassen werden (dann spricht man von Raw-Types).

  Beispiel

  ```
  List v1 = new ArrayList();
  ```

  Achtung!

  Raw-Types sollten vermieden werden, Sie wurden kurz nach der Einführung von Generics verwendet, um bestehenden Code zu migrieren.

- Wenn ein generischer Datentyp instanziiert wird, und direkt einer entsprechend getypten Variable zugewiesen wird, dann kann der konkrete Datentyp weggelassen werden (es muss aber der *Diamond Operator* <> verwendet werden).

  Beispiel

  ```
  Stack<Double> stack = new Stack<>();
  ```

  oder

  ```
  List<Integer> v1 = new ArrayList<>();
  ```

  oder

  ```
  Pair<Integer,Integer> p1 = new Pair<>(36462828, 50);
  Pair<String,Integer> p2 = new Pair<>("Michael", 2023);
  ```

Übung

2.1. Pair mit Typparametern

Erweitern Sie Ihre Klasse Pair um zwei generische Typparameter `U` und `V` für die beiden Werte, die gespeichert werden sollen.

Nutzen Sie dann die entsprechenden Typen `U` und `V` für die entsprechenden Attribute der Klasse und ggf. auch für Methodenparameter/-rückgabewerte und lokale Variablen.

Passen Sie auch die main Methode entsprechend an.

3. Eine kurz Einführung in das Java Collections Framework

Collections (d. h. Sammlungen von Objekten)

- Eine häufig benötigte Form von Datenstrukturen ist eine Collection (Sammlung), die unterschiedliche Datenelemente speichert.
  - entweder genau der gleiche Typ
  - oder der gleiche Typ; ggf. mit Subtypen
  - oder gemischte Typen (eher selten)
- Abhängig vom geplanten Gebrauch kann eine Collection…
  - schnellen Zugriff auf die einzelnen Elemente unterstützen.
  - die Sortierung der Elemente unterstützen.
  - die Möglichkeit zum Zugriff auf bestimmte Elemente geben.
  - bei Bedarf wachsen.
  - usw.

Wrapper-Klassen und Auto(un)boxing

Wiederholung

- wir unterscheiden Werte und Referenzen
- primitive Datentypen sind keine Referenztypen

  Sie werden nicht von Object abgeleitet und besitzen keine Methoden.

🔎 Beobachtung

Wie wir gesehen haben ist es möglich primitive Datentypen in Datenstrukturen wie Listen zu speichern obwohl diese eigentlich nur Objekte speichern können.

Wenn primitive Werte an Stellen verwendet werden, die eigentlich Objekte verlangen (z. B. Collections), dann werden automatisch die jeweiligen passenden Wrapperklassen verwendet; d. h. die primitiven Datentypen werden in Objekte umgewandelt und entsprechend behandelt:

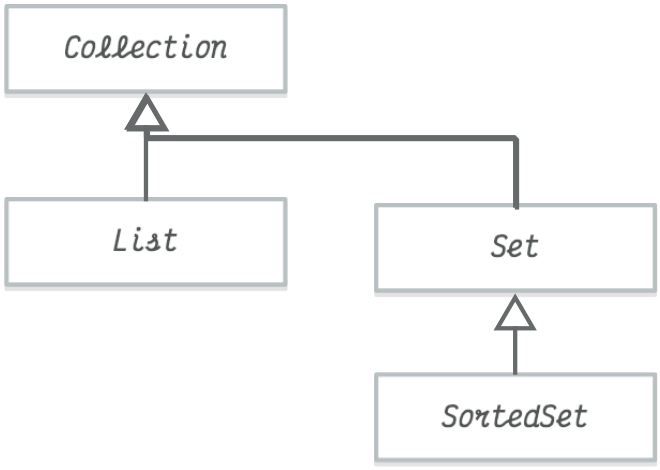
```
int -> java.lang.Integer
float -> java.lang.Float
double -> java.lang.Double
char -> java.lang.Character
boolean -> java.lang.Boolean
byte -> java.lang.Byte
short -> java.lang.Short
long -> java.lang.Long
```

Warnung

Dieses so genannten *Autoboxing* hat jedoch ggf. erhebliche Laufzeitkosten und sollte daher vermieden werden.

Grundlegende Klassen des Collections Frameworks

- Das Hauptinterface ist `java.util.Collection`
  - es definiert grundlegende Methoden für Sammlungen von Objekten.
  - es definiert keine Restriktionen / Garantien bezüglich Duplikate / Ordnung / Sortierung, usw.
- `List` (hat die Implementierungen `ArrayList`, `LinkedList`, ...)
  - Objekte sind sortiert
  - kann Duplikate enthalten
  - direkter Zugriff auf Objekte über Index
- `Set` (hat die Implementierung `HashSet`)
  - keine Einschränkung bzgl. der Sortierung
  - Objekt kann nur einmal enthalten sein
- `SortedSet` (hat die Implementierung `TreeSet`)
  - Ein Set, aber geordnet bzgl. einer spezifischen Vergleichsstrategie.



Im Folgenden betrachten wir Collections die Element vom Typ „E“ verwalten; dieser Typ ist durch „passende“ Typen ersetzbar.

java.util.List

Das Interface bietet folgende Methoden:

- `boolean add(E e)`: Anhängen des Elements `e` an die Liste
- `void add(int pos, E e)`: Einfügen des Elements `e` an Position `pos`; verschiebt alle Elemente ab Position `pos` um eine Position nach hinten
- `boolean addAll(Collection c)`: Anhängen aller Elemente der Collection `c` an die Liste
- `boolean addAll(int pos, Collection c)`: Einfügen aller Elemente der Collection `c` an Position `pos` (s.o.)
- `void clear()`: Löscht alle Elemente der Liste
- `boolean contains(Object o)`: Liefert true, wenn sich Objekt `o` in der Liste befindet
- `boolean containsAll(Collection c)`: Liefert true, falls alle Objekte der Collection `c` in der Liste sind
- `E get(int pos)`: Liefert das Element an Position `pos` der Liste
- `int indexOf(Object o)`: Liefert die erste Position, an der sich `o` in der Liste befindet, sonst -1. Gegenstück: `int lastIndexOf(Object o)`
- `boolean isEmpty()`: Liefert true wenn die Liste leer ist
- `E remove(int pos)`: Entfernt das Objekt an Position `pos` und liefert es zurück
- `boolean remove(Object O)`: Versucht Objekt `o` aus der Liste zu entfernen; true bei Erfolg
- `int size()`: Liefert die Größe der Liste
- `Object[] toArray()`: Liefert ein Array, das alle Elemente der Liste umfasst

Für Konstruktoren in den Erbenklassen gilt:

- es gibt immer Parameterlose Konstruktoren (Konvention)
- Konstruktoren mit `Collection` als Parameter kopieren alle Werte in die Liste
- Spezialfälle (siehe entsprechende Dokumentation)

Konkrete Implementierungen (Auswahl):

`java.util.LinkedList` fügt folgende Methoden hinzu (Auswahl):

- `void addFirst(E)`
- `void addLast(E)`
- `E getFirst()`
- `E getLast()`

`java.util.ArrayList` speichert die Elemente in einem Array und fügt folgende Methoden hinzu (Auswahl):

- `void ensureCapacity(int minCapacity)` - falls die Liste weniger Elemente als `minCapacity` fassen kann, wird das Array vergrößert
- `void trimToSize()` - verkleinert das Array auf die Listengröße
- `ArrayList(int initialCapacity)` Neuer Konstruktor, für die Spezifikation der Größe

java.util.Set

- Ein Set repräsentiert eine mathematische Menge

  D. h. ein gegebenes Objekt ist nur maximal einmal vorhanden und das Einfügen scheitert, wenn das Objekt schon vorhanden ist.

- Umfasst die meisten der schon bekannten Methoden

```
boolean add(E e)
boolean addAll(Collection c)
void clear()
boolean contains(Object O)
boolean containsAll(Collection c)
boolean isEmpty()
boolean remove(Object O)
boolean removeAll(Collection c)
int size()
Object[] toArray()
```

Konkrete Implementierungen (Auswahl):

- java.util.HashSet: verwaltet die Daten in einer Hashtabelle (sehr effizienter Zugriff)
- java.util.TreeSet: verwaltet die Daten in einem Baum mit Zugriffszeiten in O(log n)[1].

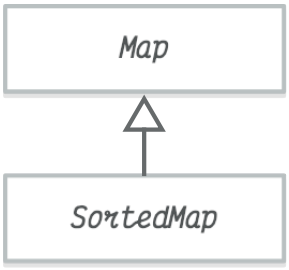
---

[1] Die Komplexität von Algorithmen diskutieren wir in einem späteren Abschnitt detailliert.

Collections für Paare von Werten (Schlüssel und Wert)

Im folgenden wird der Typ „K“ für den Typ des Schlüssels und der Typ "V" für den Typ des Wertes verwendet; diese Typen sind durch „passende“ Typen ersetzbar.

java.util.Map



Wenn Objekte nicht über einen numerischen Index, sondern über einen Schlüssel (einzigartiger, aber sonst zufälliger Wert) auffindbar sein sollen, z.B. eine Telefonnummer mit „Nachname + Vorname“.

Das Interface bietet folgende Methoden:

- `Object put(K key, V value)` speichert "value" zum Auffinden mit "key"
- `Object get(Object key)` findet das Objekt gespeichert unter "key"
- `boolean containsKey(Object key)` beantwortet, ob ein Objekt unter "key" liegt
- `boolean containsValue(Object value)` beantwortet, ob "value" in der HashMap ist
- `Object remove(Object key)` löscht "key" und die assoziierten Objekte

Wir werden später klären warum nur die Parameter der Methode `put` einen generischen Typ (`K` für *Key* (🏴 *Schlüssel*) und `V` für *Value* (🏴 *Wert*)) haben.

Konkrete Implementierungen (Auswahl):

`java.util.HashMap`

Erlaubt Zugriff auf Elemente durch einen berechneten Schlüssel, z.B. „Nachname + Vorname“ Schlüssel wird in numerischen Index (Hashwert[2]) konvertiert und für effizienten Zugriff genutzt.

[2] Hashing diskutieren wir später detailliert.

Iterieren über Collections bzw. Laufen über die Elemente eine Collection

java.util.Iterator

- Java nutzt einen `Iterator`, um über Elemente in einer Collection zu laufen („zu iterieren“).
  Normalerweise erhält man den Iterator durch den Aufruf von `iterator()` auf der Collection.
  Das gilt für alle Subklassen des Collection Interface
  Für eine `HashMap` nutzt man `keys()` und darauf `iterator()`
  `iterator()` liefert eine Instanz von `java.util.Iterator`
- Ein Iterator bietet die Operationen:
  - `boolean hasNext()` – gibt es noch weitere Elemente?
  - `Object next()` – liefert das nächste Element, falls eines existiert;
    sonst wird eine NoSuchElementException geworfen.
    Prüfen Sie vorher die Existenz mit `hasNext()`!
  - `void remove()` – entfernt das zuletzt gelieferte Element; häufig nicht unterstützt. In diesem Fall wird eine `UnsupportedOperationException` geworfen.

Beispiel

```
1 final List<Integer> l = Arrays.asList(1, 2, 3); // Liste anlegen
2 int r = 0;
3 final var it = l.iterator(); // Iterator holen
4 while(it.hasNext()) // weiter, solange Elemente da
5     r += it.next(); // Element zur Summe addieren
```

Sollte aufgrund von Domänenwissen bekannt sein, dass die Liste niemals leer ist, kann die Schleife auch so geschrieben werden:

```
1 do {
2     r += it.next(); // Element zur Summe addieren
3 } while(it.hasNext()); // weiter, solange Elemente da
```

Weiterhin gibt es eine besondere `for`-Schleife (*foreach-loop*), die die Iteration über eine `Collection`, die das Interface `Iterable` implementiert vereinfacht:

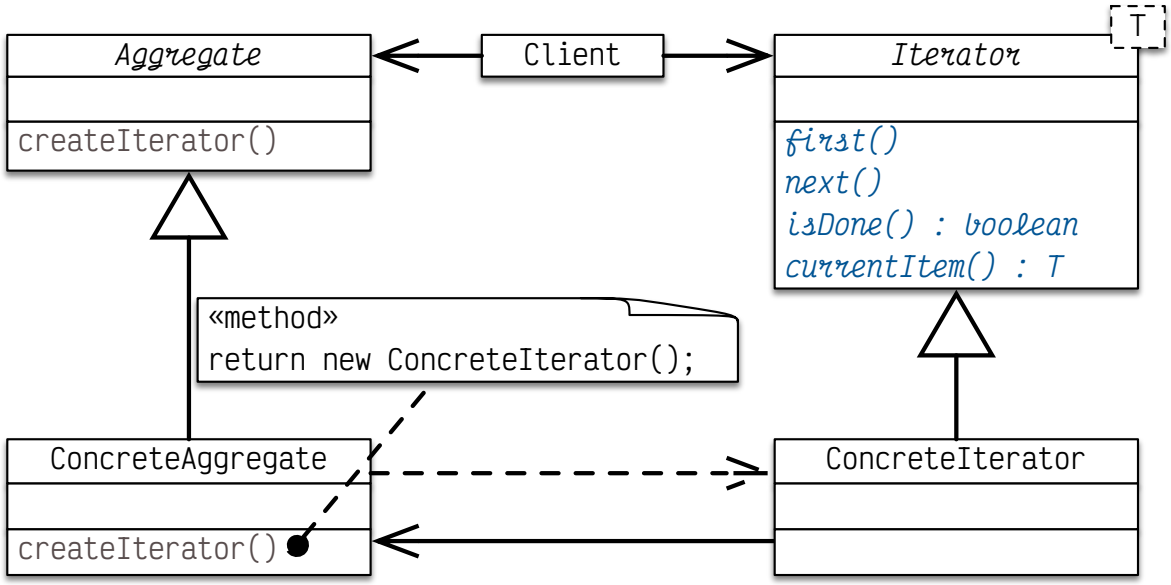
Beispiel

```
1 for (Integer i : l) // für jedes Element in der Liste
2     r += i; // Element zur Summe addieren
```

`java.util.Iterable` definiert dabei lediglich das Protokoll zur Erzeugung eines `java.util.Iterator`s.

Das Iterator-Design Pattern

Die Implementation von *Iterator*s ist ein Beispiel für die Umsetzung des *Design Pattern* ( *Entwurfsmuster*) „Iterator“.



Es ist hier festzustellen, dass in Java die Methode `hasNext()` an die Stelle der Methode `isDone()` rückt. die Methode `next()` das Fortschalten des Iterators und die Rückgabe des nächsten Elements kombiniert. In anderen Sprachen bzw. im Textbuch sind diese beiden Operationen getrennt. Eine Design Pattern stellt auch immer nur eine Blaupause dar, die in der konkreten Umsetzung angepasst werden kann bzw. soll.

Übung

3.1. Iterables

Implementieren Sie das Interface `java.lang.Iterable` für Ihre Klasse `Pair`. D. h. schreiben Sie eine Methode `java.util.Iterator iterator()`, die einen Iterator für die Elemente des Paares zurückgibt.

Dazu ist es erforderlich, dass Sie eine Klasse `PairIterator` implementieren, die das Interface `java.util.Iterator` implementiert. Diese Klasse führt dann die eigentliche Iteration durch. Die Erzeugung der Instanz von `PairIterator` erfolgt in der Methode `iterator()`.

Hinweis

Die Klasse `PairIterator` benötigt einen Konstruktur, der eine Referenz auf das Pair bekommt, über das iteriert werden soll.

4. Generics - Fortgeschrittene Konzepte

Typkompatibilität

```
1 List<String> ls = new LinkedList<String>();
2 List<Object> lo = ls;
3 lo.add(new Object());
4 String s = ls.get(0);
```

? Frage

Wo können hier Probleme auftreten?

✓ Antwort

Die Zuweisung in Zeile 2 ist nicht erlaubt, da `List<String>` und `List<Object>` nicht kompatibel sind. Obwohl String ein Subtype von Object ist, ist `List<String>` kein Subtyp von `List<Object>`. Wäre dies erlaubt, dann könnte man in Zeile 3 ein Objekt vom Typ `Object` einer Liste von Strings hinzufügen!

Zusammenfassung

Generics sind in Java *invariant*.

Wildcards

? Frage

Wie können wir eine Methode schreiben, die auch mit Subtypen von generischen Typen arbeiten kann?

Eine einfache Methode zum Ausgeben eines Stacks:

```
static void printAll(Stack<Object> stack) {
    for (int i = 0; i < stack.size(); i++) {
        System.out.print(stack.get(i) + " ");
    }
    System.out.println();
}
```

Diese Methode definiert einen Parameter vom Typ `Stack<Object>`. Das bedeutet, dass nur `Stack<Object>`-Objekte übergeben werden können.

Die Implementierung funktioniert aber auch mit Listen von Subtypen von `Object` wie `String` oder `Integer`. Ein Aufruf mit einem `Stack<Integer>`-Objekt führt zu einem Compilerfehler:

```
printAll(new Stack<Integer>())
|  Error:
|  incompatible types:
|        Stack<java.lang.Integer>
|        cannot be converted to
|        Stack<java.lang.String>
```

Eine Lösung ist die Verwendung von Wildcards:

```
static void printAll(Stack<?> stack) {
    for (int i = 0; i < stack.size(); i++) {
        System.out.print(stack.get(i) + " ");
    }
    System.out.println();
}
```

Durch die Verwendung von `Stack<?>` kann die Methode mit allen Subtypen von `Object` aufgerufen werden.

Achtung!

Durch die Verwendung eines Wildcards ist es nicht mehr möglich Elemente hinzuzufügen, da der konkrete Typ des Stacks nicht bekannt ist.

```
Stack<?> stack = new Stack<Integer>();
stack.add(1); // Compilerfehler
```

Beispiel

```
        Stack<String> infix = new Stack<>();
        Stack<Double> ops = new Stack<>();
        for (String arg : args) {
            switch (arg) {
                case "+":
                    ops.push(ops.pop() + ops.pop());
                    infix.push("(" + infix.pop() + " + " + infix.pop() + ")");
                    break;
                case "*":
                    ops.push(ops.pop() * ops.pop());
                    infix.push("(" + infix.pop() + " * " + infix.pop() + ")");
                    break;
                default:
                    infix.push(arg);
                    ops.push(Double.parseDouble(arg));
            }
```

```
17          }
```

Beschränkte Wildcards

```
1  abstract class Shape {
2  abstract void draw(Canvas c);
3  }
4  class Circle extends Shape {
5  void draw(Canvas c) { /*...*/ }
6  }
7  class Rectangle extends Shape {
8  void draw(Canvas c) { /*...*/ }
9  }
10 class Canvas {
11 void draw(Shape s) {
12     s.draw(this);
13 } }
```

Aufgabe: Definition einer Methode `drawAll(<list of shapes>)` für `Canvas`, die eine Liste von Formen zeichnet?

! `drawAll(List<Shape> shapes)` würde nur mit Listen von `Shape`-Objekten funktionieren.

! `drawAll(List<?> shapes)` würde alle Listen von `Shape`-Objekten und allen Subtypen von `Shape` funktionieren, aber auch Listen von anderen Typen.

Wir müssen den Type der Liste auf `Shape` und Subtypen von `Shape` beschränken. Dies erreichen wir mit einem *beschränkten Wildcard*:

`drawAll(List<? extends Shape> shapes)` funktioniert mit Listen von `Shape`-Objekten und allen Subtypen von `Shape`.

`? extends X` bedeutet:

- Wir kennen den exakten Typ nicht („?“)
- Aber wir wissen, dass der Typ zu `X` konform sein muss
- `X` ist die „obere Schranke“ der Wildcard

Wo ist das Problem bei folgender Methode?

```
1  void addRectangle(List<? extends Shape> shapes) {
2      shapes.add(0, new Rectangle());
3  }
```

Das Problem ist, dass wir nicht wissen, welcher konkrete Typ von `List` übergeben wird. Es könnte auch eine `List<Circle>` sein, die keine Rechtecke aufnehmen kann.

Die Methode `addRectangle` würde also nicht mit einer `List<Circle>` funktionieren.
Die Lösung ist die Spezifikation einer unteren Schranke. Dies ist mittels der Verwendung von *super* möglich.

```
1  void addRectangle(List<? super Rectangle> shapes) {
2      shapes.add(0, new Rectangle());
3  }
```

`? super X` bedeutet:

- Wir kennen den exakten Typ nicht („?“)
- Aber wir wissen, dass `X` von dem unbekannten Typ abgeleitet sein muss
- `X` ist die „untere Schranke“ der Wildcard

Generische Methoden

Zusätzlich zu generischen Klassen können auch generische Methoden definiert werden. Bei diesen Methoden wird der generische Typ vor dem Rückgabetyp spezifiziert.

Beispiel

```java
public class Tuple2<T> {
    public final T first;
    public final T second;
    public Tuple2(T first, T second) {
        this.first = first; this.second = second;
    }
    public String toString() { return "(" + first + ", " + second + ")"; }

    public <X> Tuple2<X> map(Function<T,X> mapper) {
        return new Tuple2<>(mapper.apply(first), mapper.apply(second));
    }
}
```

Verwendung

```java
/*Tuple2<String> ts =*/ new Tuple2<Integer>(1,2).map(e -> "value: " + e)
```

Statische Typisierung

- Statische Typsysteme sind (noch immer) Gegenstand der Forschung
- Java-ähnliche Typsysteme sind begrenzt, aber im Allgemeinen können Typsysteme sehr mächtig und ausdrucksstark sein - aber auch sehr kompliziert
- Manche Programmierer sehen statische Typsysteme als eine Begrenzung ihrer Freiheit (*„ich weiß, was ich tue“*)
- Andere Programmierer denken, dass statische Typsysteme nicht nur viele Fehler erkennen, sondern auch eine gute Struktur im Code erzwingen (*„erst denken, dann schreiben“*)
- In der Praxis zeigt sich, dass fast alle großen Projekte auf statische Typsysteme setzen.

Übung

4.1. Wildcards

1. Fügen Sie Ihrer generischen Klasse `Pair` eine Methode `addToMap(...)` hinzu, die die Elemente des Pairs in einer `java.util.Map` speichert. D. h. der erste Wert eines Pairs wird als Schlüssel und der Zweite als Value verwendet.

   Achten Sie darauf, dass die Methode auch Maps von Supertypen von `U` und `V` akzeptiert. D. h. es sollte folgendes Szenario unterstützt werden:

   ```
   1 Pair<Integer,Integer> p = new Pair<>(1, 2);
   2 java.util.Map<Object,Integer> map = new java.util.HashMap<>();
   3 p.addToMap(map); // D.h. dem Key "1" ist nur der Wert "2" zugewiesen.
   ```

2. Schreiben Sie eine Methode die die Werte eine `Pair`s aktualisiert basierend auf den Werten eines anderen Paares. Achten Sie darauf, dass die Methode auch mit Subtypen von `U` und `V` arbeitet.

   ```
   1 Pair<Integer,Integer> pIntegers = new Pair<>(1, 2);
   2 Pair<Object,Object> pObjects = new Pair<>("a",new Object());
   3 pObjects.update(pIntegers);
   ```