252-0027
Einführung in die Programmierung

15. Generics

Manuela Fischer, Malte Schwerhoff

Departement Informatik
ETH Zürich

Einstieg

- Bisher gesehen
  1. *Implementation* verkettete List – aber nur für *einen* Typ (`int`)
  2. *Nutzung* verschiedener *generischer* Collection-Klassen, z.b. `ArrayList<E>` für beliebigen Typ `E`
- Neu in diesem Kapitel:
  1. Einfache generische Klassen selbst implementieren
  2. Teaser fortgeschrittene Nutzung von Generics

Falls Sie tief in Generics eintauchen möchten: https://angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html

15. Generics

15.1 Generische Klassen selbst entwickeln

15.2 Type Erasure

15.3 Teaser: Generics für Fortgeschrittene

Ziel von Generics: Generische Klassen

- **Ähnliches Ziel:** Generische **Methoden**, für beliebige Typen aufrufbar
  - `println(x)` — für `x` jeden Typs
  - `Arrays.fill(xs, …)` — für jedes Array `xs`
- **Lösung: Overloading**
  - Eine Methode pro primitivem Typ
  - Plus eine für `Object`

```
println()
println(boolean x)
println(char x)
println(char[] x)
println(double x)
println(float x)
println(int x)
println(long x)
println(Object x)
println(String x)
```

```
fill(boolean[] a, boolean val)
fill(byte[] a, byte val)
fill(char[] a, char val)
fill(double[] a, double val)
fill(float[] a, float val)
fill(int[] a, int val)
fill(long[] a, long val)
fill(short[] a, short val)
fill(Object[] a, Object val)
```

Ziel von Generics: Generische Klassen

- **Ziel: Generische Klasse**, für beliebige Typen instanziierbar

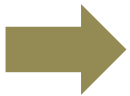
```
public class Pair {
  ? first;
  ? second;

  …
}
```

```
new Pair(1, 2);
new Pair(true, false);
new Pair("links", "rechts");
…
```

- **Problem:** Attribute können *nicht* überladen werden
  - In Java (allen Sprachen?) nicht möglich (technisch sinnvolle Umsetzung unklar)
- **Lösung:** Typparameter (in Java und anderen Sprachen)

Beispiel: Klasse mit Generics versehen

```
public class StringPair {
  private String first;
  private String second;

  public StringPair(String first,
                    String second) {

    this.first = first;
    this.second = second;
  }

  public String getFirst() {
    return this.first;
  }

  public String toString() {
    return "(" + this.first + ","
               + this.second + ")";
  }

  // … z.B. setFirst(), getSecond()
}
```



```
public class Pair<E> {
  private E first;
  private E second;

  public Pair(E first,
              E second) {

    this.first = first;
    this.second = second;
  }

  public E getFirst() {
    return this.first;
  }

  public String toString() {
    return "(" + this.first + ","
               + this.second + ")";
  }

  // … z.B. setFirst(), getSecond()
}
```

Allgemein: Generische Klasse entwickeln

Empfohlenes Vorgehen (mindestens für den Anfang)

1. Überlegen: Muss Code wirklich typ-generisch sein?
2. Nicht-generische Version entwickeln – z.B. für `String` oder `Integer` – und testen
3. Code generalisieren
    1. *`Klasse<E>`* statt nur *`Klasse`*
    2. Bisherigen festen Typ (z.B. `String`) an allen *relevanten* Stellen durch **Typvariable** (**Typparameter**) *E* ersetzen
        - Name für Typvariable frei wählbar – z.B. `E`, `T`, `Elem`
        - Konvention: Beginnen mit einem (bzw. sind) Grossbuchstaben

```
public class StringPair {
  private String first;
  …
}
```

```
public class Pair<E> {
  private E first;
  …
}
```

Beispiel: Generische Klasse nutzen

```
public class Pair<E> {
  private E first;
  private E second;

  public Pair(E first,
              E second) {

    this.first = first;
    this.second = second;
  }

  public E getFirst() {
    return this.first;
  }

  …
}
```

Deklaration

```
Pair<String> p1;
```

… und Initialisierung

```
Pair<Integer> p2 = new Pair<Integer>(1, 2);
…println(p2.getFirst()); // 1
```

… und Initialisierung mit Typinferenz

```
Pair<String> p3 = new Pair<>("one", "two");
…println(p3.getSecond()); // two
```

Beispiel: Paar mit zwei Typparametern

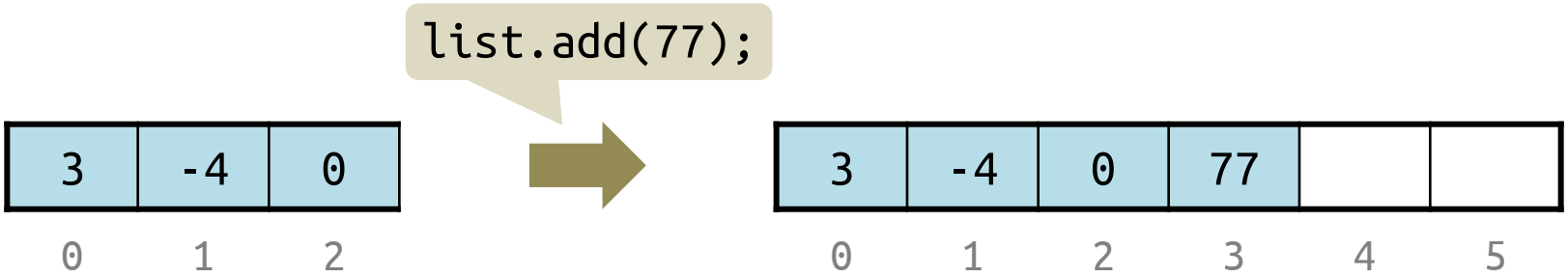
```
public class Pair<E1, E2> {
  private E1 first;
  private E2 second;

  public Pair(E1 first,
              E2 second) {

    this.first = first;
    this.second = second;
  }

  public E1 getFirst() {
    return this.first;
  }

  public E2 getSecond() {
    return this.second;
  }

  …
}
```

```
Pair<Integer, String> p1 =
  new Pair<>(137, "Einhundertsiebenunddreissig");
```

```
Pair<Employee, Random> p2 =
  new Pair<Employee, Random>(
    new Employee(…),
    new Random());

int bonus =
  p2.getSecond().nextInt(0, p2.getFirst().getSalary());
```

Grösseres Beispiel: ArrayList<E>

- Erinnerung: ArrayList<E> ist ein «wachsendes Array»
  - `int[] array = new int[3];`
    — Grösse des Arrays (hier: 3) kann nicht mehr verändert werden
  - `ArrayList<Integer> list = new ArrayList<Integer>();`
    — Elemente Hinzufügen vergrössert die Liste nach und nach



Grösseres Beispiel: ArrayList<E>

- **Ziel: ArrayList<E> selbst implementieren**
- **Mindestfunktionalität**

`ArrayList()`	– Konstruktor einer leeren Liste
`int size()`	– Gibt die Anzahl der Elemente zurück
`E get(int idx)`	– Gibt das Element am Index zurück
`void set(int idx, E elem)`	– Ersetzt das Element am Index
`void add(E elem)`	– Hängt ein Element an
`E remove()`	– Entfernt das letzte Element
`String toString()`	– Liefert die textuelle Repräsentation

ArrayList: Klasse, Attribute, Konstruktor

```
public class ArrayList<E> {
  private static final int DEFAULT_CAPACITY = 10;

  private E[] elements; // INV: != null
  private int size; // INV: 0 <= size <= elements.length

  public ArrayList() {
    elements = (E[]) new Object[DEFAULT_CAPACITY];
    size = 0;
  }

  …
}
```

leere Liste gdw.
`size == 0`

new E[ ] nicht erlaubt
(Details folgen noch)

ArrayList: `size()`

```
public class ArrayList<E> {
  …
  private int size;
  …

  public int size() {
    return size;
  }
}
```

ArrayList: `get()` …

```
class ArrayList<E> {
  E[] elements;
  int size;
  …
}
```

```
// In class ArrayList
public E get(int index) {
  if (index < 0) {
    throw new IllegalArgumentException("Index negative");
  } else if (size <= index) {
    throw new IllegalArgumentException("Index exceeds size");
  }

  return elements[index];
}
```

Indexüberprüfung kann auch an `java.util.Objects.checkIndex(index, size)` delegiert werden

ArrayList: … und set()

```
class ArrayList<E> {
  E[] elements;
  int size;
  …
}
```

```
// In class ArrayList
public void set(int index, E element) {
  if (index < 0) {
    throw new IllegalArgumentException("Index negative");
  } else if (size <= index) {
    throw new IllegalArgumentException("Index exceeds size");
  }

  elements[index] = element;
}
```

Einschub: Dokumentation

- Methoden sollten stets dokumentiert werden
  - Insbesondere: Klient *muss* Index-Anforderungen bei `get()`/`set()` kennen
- Mindestens zwei Möglichkeiten
  1. Im Javadoc-Format (Industriestandard)

```
20  /**
21   * Returns the element at the specified position in this list.
22   *
23   * @param index index of the element to return
24   * @return the element at the specified position in this list
25   * @throws IndexOutOfBoundsException if the index is out of range
26   *         ({@code index < 0 || index >= size()})
27   */
28  public E get(int index) {
29    if (index < 0) throw new IllegalArgumentException("Index cannot b
```

get

```
public E get(int index)
```

Returns the element at the specified position in this list.

Parameters:

`index` - index of the element to return

Returns:

the element at the specified position in this list

Throws:

`IndexOutOfBoundsException` - if the index is out of range (`index < 0 || index >= size()`)

Javas List<E>.get(): Quellcode und daraus erzeugte API-Dokumentation

Einschub: Dokumentation

2. Selbstgewähltes Format, z.B.

```
35    // PRE: 0 <= index <= size()
36    // POST: get(index) == element
37    public void set(int index, E element) {
38        if (index < 0) throw new IllegalArgumentException("Index cannot be negative");
```

- Langfristig: Klare Empfehlung für JavaDoc, da Standard
- In EProg (leider) kein Lernziel, aber
  - Dokumentieren zwingt einen dazu, sich über die Rolle/den Nutzen einer Methode gedanken zu machen
  - Wenn Sie bereits jetzt dokumentieren, müssen Sie sich später nicht umgewöhnen

ArrayList: add()

```
class ArrayList<E> {
  E[] elements;
  int size;
  …
}
```

```
// In class ArrayList
public void add(E element) {
  if (size >= elements.length) {
    grow();
  }
  elements[size] = element;
  size++;
}

private void grow() {
  int newCapacity = elements.length * 2;
  elements =
    Arrays.copyOf(elements, newCapacity);
}
```

- Array voll → Kapazität verdoppeln
  - `java.util.Arrays.copyOf()` kopiert Elemente in ein neues Array
- Verdoppelung → amortisierte Laufzeitkosten $\mathcal{O}(1)$
  - Auch andere Wachstumsstrategien möglich, z.B. mal 1,5 (OpenJDK)
- Hier ignoriert: `newCapacity < 0` (numerischer Überlauf)

Quellcode Javas ArrayList<E>.grow()

ArrayList: `toString()`

```
class ArrayList<E> {
  E[] elements;
  int size;
  …
}
```

```
// In class ArrayList
public String toString() {
  return Arrays.toString(elements);
}
```

Sinnvolle Implementation? 🤔

ArrayList: `toString()`

```
class ArrayList<E> {
  E[] elements;
  int size;
  …
}
```

```
// In class ArrayList
public String toString() {
  if (size == 0) {
    return "[]";
  } else {
    String result = "[" + elements[0];

    for (int i = 1; i < size; i++) {
      result += ", " + elements[i];
    }

    return result + "]";
  }
}
```

Iteration bis `size`, nicht bis `elements.length` (Kapazität)

- Andernfalls würden auch ungenutzte Array-Positionen (`null`) ausgegeben

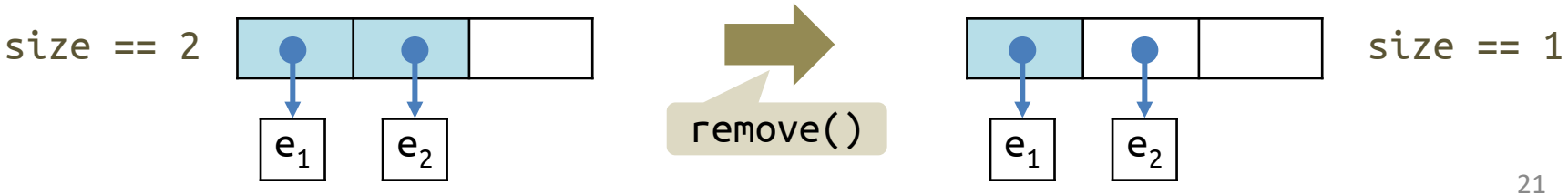
In der Praxis empfohlen, da effizienter: Rückgabe via `StringBuilder` (statt direkt via `String`) zusammensetzen

ArrayList: `remove()`

```
class ArrayList<E> {
  E[] elements;
  int size;
  …
}
```

```
// In class ArrayList
public E remove() {
  if (size == 0) {
    throw new NoSuchElementException();
  }

  E last = elements[size - 1];
  size--;

  return last;
}
```

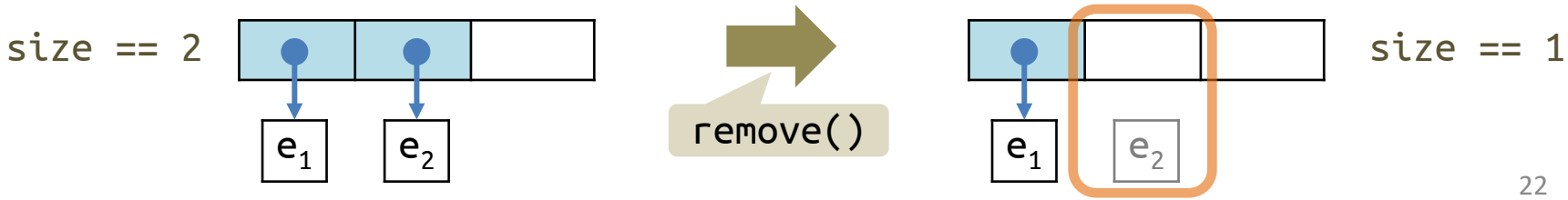
Ausreichend? 🤔



ArrayList: `remove()`

```
// In class ArrayList
public E remove() {
  if (size == 0) { … }

  E last = elements[size - 1];
  elements[size - 1] = null;
  size--;

  return last;
}
```

- `elements[…] = null` bedeutet «entferne Element aus der Liste»
  - Nötig, damit das Element, falls nirgends mehr in Gebrauch, vom Garbage Collector als «totes» Objekt erkannt werden kann
  - Kein `null`-Setzen ➜ Element weiterhin vom Array aus erreichbar und somit nicht «tot»
- In der Praxis wichtig; für EProg nur Randnotiz



Ideen für weitere Methoden

- `ArrayList(int capacity)` – Leere Liste mit initialer Kapazität
  - 🤔 Delegieren des bisherigen Konstruktors `ArrayList()` auf neuen Konstruktor?

- `void add(int index, E element)` – Fügt neues Element am Index ein
- `E remove(int index)` – Entfernt Element am Index
  - 🤔 Effekt auf Elemente vor bzw. hinter dem betroffenen Index?

- `int indexOf(E element)` – Gibt den Index des Elements zurück
- `boolean remove(E element)` – Entfernt das Element
  - 🤔 Element nicht vorhanden? Mehrfach vorhanden?

15. Generics

15.1 Generische Klassen selbst entwickeln

15.2 Type Erasure

15.3 Teaser: Generics für Fortgeschrittene

Rückblick: Generics in Java

Generics wurden mit Java 5 zur Sprache hinzugefügt – acht Jahre nach dem ersten Release von Java

- Erweiterung einer Sprache erfordert Abwägen verschiedener Aspekte: z.B. Schwierigkeit der Implementation, Zusammenspiel mit anderen Sprachfeatures, Laufzeitkosten
- Typische Wahlmöglichkeiten bei der Weiterentwicklung einer Sprache
  1. Rückwärtskompatibilität: Existierender Code ist auch mit neueren Versionen kompatibel
     - Schränkt den Spielraum der Sprachdesigner/innen ein bzw. erfordert (unschöne) Kompromisse
     - Kein Aufwand für Nutzer/innen von Java: Programme funktionieren einfach weiter
  2. Nicht rückwärtskompatibel (→ Vor-/Nachteile andersherum)
- Java wird grösstenteils rückwärtskompatibel weiterentwickelt → führt mitunter zu notwendigen aber unschönen Kompromissen

Type Erasure («Typlöschung»)

- Java 4: Java und die JVM kannten keine Generics
- Java 5: Rückwärtskompatibel um Generics erweitert
  - Generics stehen neu dem Compiler und Typechecker zur Verfügung …
  - … aber der JVM-Bytecode kennt weiterhin keine Generics
    - Generics werden nach Typechecking quasi durch `Object` ersetzt
    - JVM muss generische Typen daher nicht unterstützen
    - Generischer Typ zur Laufzeit daher nicht mehr vorhanden

Java 5: Generische Klassen in Java …

```
public class ArrayList<E> {
  private E[] elements;
  …
  E get(int i);
  void set(int i, E e);
}
```

… aber nicht im JVM-Bytecode: konzeptionell wird `E` durch `Object` ersetzt

Java 4: Keine generischen Klassen

```
public class ArrayList {
  private Object[] elements;
  …
  Object get(int i);
  void set(int i, Object e);
}
```

Type Erasure: Konsequenzen Overloading

```
void compute(ArrayList<String> data) {
  …println("Got strings …");
}

void compute(ArrayList<Integer> data) {
  …println("Got integers …");
}
```

Compilerfehler, da beide Methoden nach Type-Erasure die gleiche Signatur haben, d.h. nicht mehr unterscheidbar sind:

```
void compute(ArrayList<Object> data)
```

Type Erasure: Konsequenzen (Auszug)

```
// In class ArrayList<E>
new E() // Compilerfehler
```

Da Typ E zur Laufzeit unbekannt sein wird

```
// In class ArrayList<E>
new E[n] // Compilerfehler
// Stattdessen: (E[]) new Object[n]
```

Analog

(und weil Arrays primitiver als Klassen sind)

```
ArrayList<String> xs = new ArrayList<>();
ArrayList<Integer> ys = new ArrayList<>();
…println(xs.equals(ys)); // true
```

`true` da zur Laufzeit beide quasi `ArrayList<Object>`

(aber auch nur weil beide Listen leer sind)

Type Erasure: Abschluss

Für EProg sollten Sie wissen, dass

- Es Type Erasure gibt
- Dies zu Problemen/Überraschungen führen kann

15. Generics

Generische Typen und Subtyping

1. Gegeben zwei Typen A und B, mit B Subtyp A
   - Erzeugt z.B. durch `class B extends A` (oder `class B implements A`)
2. Gegeben ein generischer Typ X, z.B. `ArrayList<·>`, aber auch Arrays
3. Frage: In welcher Beziehung sollten X<B> und X<A> stehen?
   - X<B> Subtyp X<A>? → kovariant (Kovarianz)
   - X<A> Subtyp X<B>? → kontravariant (Kontravarianz)
   - Keine? → invariant (Invarianz)

Warum ist das wichtig?

```
void compute(Object[] data) {
  …
}
```

```
Integer[] data =
  new Integer[8];

compute(data);
```

```
void compute(ArrayList<Object> data) {
  …
}
```

```
ArrayList<Integer> data =
  new ArrayList<Integer>();

compute(data);
```

Sollten diese Aufrufe erlaubt sein?

Kovarianz: Mögliches Problem

- Kovariant: Falls B Subtyp A, dann X<B> Subtyp X<A>
- Hier: `Manager` (= B) Subtyp `Employee` (= A)

```
void compute(Employee[] staff) {
  staff[0] = new Employee("Muriel");
}
```

```
Manager[] managers = new Manager[] {
  new Manager("Astrit", 127),
  …
};

compute(managers);

for (Manager m : managers) { …println(m.managementLevel); }
```

Laufzeitfehler: `Employee` hat keine Attribut `managementLevel`

Kontravarianz: Mögliches Problem

- Kontravariant: Falls B Subtyp A, dann X<A> Subtyp X<B>
- Hier: `Manager` (= B) Subtyp `Employee` (= A)

```
void compute(Manager[] managers) {
  for (Manager m : managers) {
    …println(m.managementLevel); }
}
```

Laufzeitfehler: `Employee` hat keine Attribut `managementLevel`

```
Employee[] staff = new Employee[] {
  new Employee("Muriel"),
  …
};

compute(data);
```

Lösung in der Theorie (vereinfacht)

- **Invariant** (X<A> und X<B> in keiner Beziehung) ist immer sicher
- X< ·> darf **kovariant** sein, wenn (nach der Initialisierung) nur Lesezugriffe erfolgen können erfolgen
  - Anwendungsbeispiel: unveränderliche («immutable») Collections
- X< ·> darf **kontravariant** sein, wenn nach Upcast nur Lesezugriffe erfolgen
  - Anwendungsbeispiel: Serializer (konvertiert ein Objekt in ein Text- oder Byte-Repräsentation, z.B. zwecks Verschickens über ein Netzwerk)
- Arrays (schreib- und lesbar) sollten also invariant sein

Arrays in Java: Kovariant

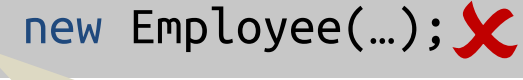
- **Arrays** sind **kovariant** (nicht invariant)
- Gründe
  1. Funktionen wie z.B.
     - `copy(Object[] source, Object[] dest)`
     - `shuffle(Object[] data)`

     könnten sonst nicht für `String[]` usw. aufgerufen werden und müssten für jeden Typ neu implementiert werden
  2. Arrays existierten von Anfang an, Generics kamen erst später dazu

Arrays in Java: Kovariant mit Laufzeitchecks

- **Arrays** sind **kovariant** – aber mit Laufzeitchecks versehen
- Problematische *Schreibzugriffe* werden zur *Laufzeit* verhindert

```
Employee[] staff = new Manager[] { … };
staff[0] = new Employee(…);
```



Laufzeitfehler:
`java.lang.ArrayStoreException`

Generische Typen in Java: Invariant

- **Generische Typen** (ArrayList, …, ausser Arrays) sind **invariant**

```
ArrayList<Employee> list1 = new ArrayList<Manager>();
```

```
ArrayList<Manager> list2 = new ArrayList<Employee>();
```

Compilerfehler
(beide Zuweisungen nicht erlaubt)

Methodensignaturen und Varianz

- Analoge Frage für Methoden: Wann ist Methodensignatur `A foo(B)` ein Subtyp von `C foo(D)`?
- Relevant für
  - Methodenüberschreibung («method overriding»)
    - In Java: Invariant in den Parametertypen, kovariant im Rückgabetyp
    - Vertiefen wir nicht weiter
  - Lambda-Funktionen, Funktionen höherer Ordnung
    - In Java verfügbar; Thema der Vorlesung *Funktionale Programmierung*

Varianz: Abschluss

- Legen Beziehung zwischen Instanzen generischer Klassen fest
  - Wann ist X<B> ein Subtyp von X<A>?
- Für EProg relevant: Verhalten von Arrays
  - Falls B Subtyp A, dann auch B[ ] Subtyp A[ ]
    (d.h. Java-Arrays sind kovariant)
- Andere Sprachen haben andere Entscheidungen getroffen
  - Trade-off
    - Sicherheit ⟷ Flexibilität
    - Statische Kosten (= Typsystemkomplexität) ⟷ dynamische Kosten (= Laufzeitchecks)

15. Generics

Generics und Subtyping: Obere Schranken

Ziel: Methode `as.addAllFrom(bs)`, die alle Elemente aus `ArrayList bs` zur `ArrayList as` hinzufügt

```
ArrayList<Object> xs = …;
ArrayList<String> ys = …;
ArrayList<Integer> zs = …;
```

```
xs.addAllFrom(ys);
ys.addAllFrom(xs);
xs.addAllFrom(zs);
ys.addAllFrom(zs);
```

Welche Aufrufe sollten erlaubt sein?

```
public class ArrayList<E> {
  private E[] elements;
  …
  public void addAllFrom(ArrayList<???> other) {
    …
  }
}
```

Welche Signatur braucht es hier?

Generics und Subtyping: Obere Schranken

Ziel: Methode `as.addAllFrom(bs)` erlaubt das Hinzufügen,
falls der Typ der Elemente in `bs` ein *Subtyp* der Elemente in `as` ist.

Andernfalls: Compilerfehler.

```
ArrayList<Object> xs = …;
ArrayList<String> ys = …;
ArrayList<Integer> zs = …;
```

```
xs.addAllFrom(ys);
ys.addAllFrom(xs);   ✗
xs.addAllFrom(zs);
ys.addAllFrom(zs);   ✗
```

Obere Typschranken

```
ArrayList<Object> xs = …;
ArrayList<String> ys = …;
ArrayList<Integer> zs = …;
```

Ziel: Methode `as.addAllFrom(bs)` erlaubt das Hinzufügen, falls die `bs`-Elemente ein *Subtyp* der `as`-Elemente sind.

```
xs.addAllFrom(ys);
ys.addAllFrom(xs);
xs.addAllFrom(zs);
ys.addAllFrom(zs);
```

1. Umsetzungsmöglichkeit: Wildcard extends E

```
// In class ArrayList<E>
public void addAllFrom(ArrayList<? extends E> other) {
  for (Object e : other.elements) {
    add((E) e);
  }
}
```

«Wildcard» (anonyme Typvariable)

Elemente in `other` müssen Subtyp von E sein

Wildcard → keine Typvariable zur Hand
→ müssen mit `Object` und Cast nach E arbeiten

Obere Typschranken

```
ArrayList<Object> xs = …;
ArrayList<String> ys = …;
ArrayList<Integer> zs = …;
```

Ziel: Methode `as.addAllFrom(bs)` erlaubt das Hinzufügen, falls die `bs`-Elemente ein *Subtyp* der `as`-Elemente sind.

```
xs.addAllFrom(ys);
ys.addAllFrom(xs);
xs.addAllFrom(zs);
ys.addAllFrom(zs);
```

2. Umsetzungsmöglichkeit: Methoden-Typparameter extends E

```
// In class ArrayList<E>
public <E1 extends E> void addAllFrom(ArrayList<E1> other) {
  for (E1 e : other.elements) {
    add(e);
  }
}
```

Typvariable, deren Gültigkeitsbereich nur diese eine Methode ist

Generics und Subtyping: Untere Schranke

Ziel: Methode `as.addAllTo(bs)`, die alle Elemente aus `ArrayList as` zur `ArrayList bs` hinzufügt (Richtung invertiert relativ zu `addAllFrom(…)`)

```
ArrayList<Object> xs = …;
ArrayList<String> ys = …;
ArrayList<Integer> zs = …;
```

```
xs.addAllTo(ys);
ys.addAllTo(xs);
xs.addAllTo(zs);
ys.addAllTo(zs);
zs.addAllTo(xs);
```

Welche Aufrufe sollten erlaubt sein?

```
public class ArrayList<E> {
  private E[] elements;
  …
  public void addAllTo(ArrayList<???> other) {
    …
  }
}
```

Welche Signatur braucht es hier?

Untere Typschranken

```
ArrayList<Object> xs = …;
ArrayList<String> ys = …;
ArrayList<Integer> zs = …;
```

Ziel: Methode `as.addAllTo(bs)` erlaubt das Hinzufügen, falls die `as`-Elemente ein *Supertyp* der `bs`-Elemente sind.

```
xs.addAllTo(ys);
ys.addAllTo(xs);
xs.addAllTo(zs);
ys.addAllTo(zs);
zs.addAllTo(xs);
```

Umsetzungsmöglichkeit: Wildcard super E

```
// In class ArrayList<E>
public void addAllTo(ArrayList<? super E> other) {
  for (E e : elements) {
    other.add(e);
  }
}
```

Elemente in `other` müssen Supertyp von E sein

Typschranken: Abschluss

- Typschranken («type bounds»)
  - Erhöhen die Flexibilität des Typsystems
  - Erhöhen so das Potenzial für Code-Wiederverwendung
- Für EProg nicht weiter relevant
  - Aber Sie sehen solche Typschranken eventuell, z.B. in der Dokumentation der `ArrayList`

```
boolean                    addAll(int index, Collection<? extends E> c)
```

```
boolean                    removeIf(Predicate<? super E> filter)
```