Check und Prepare

Lösungsvorschläge

TECHNISCHE UNIVERSITÄT DARMSTADT

Autoren: Nhan Huynh und Darya Nikitina
Fachbereich: Informatik
Übungsblatt: 08
Version: 20. November 2021
Semesterübergreifend

V1 Theoriefragen

★★★

1. Welche Vorteile ergeben sich durch die Nutzung von Generics?
2. Diskutieren Sie Vor- und Nachteile von Collections gegenüber Arrays unter den folgenden Aspekten:
   (a) Finden von Elementen
   (b) Einfügen neuer Elemente
   (c) Löschen von Elementen
3. Auch in Racket haben Sie Listen kennengelernt. Ist das Java-Interface
4. `java.util.List` vergleichbar mit den Listen aus Racket?

Lösungsvorschlag:

1. Die Verwendung von Generics ermöglicht es weniger redundanten Code zu schreiben, da man eine Klasse/Methode durch Generizität auf mehrere Referenztypen anwenden kann, ohne diese immer wieder neu speziell für einen Typ schreiben zu müssen. Ein Beispiel dafür ist das Interface
2. `java.util.Collection` in Java.
3. (a) Ein Array bietet zwei Möglichkeiten nach Elementen zu suchen:
      i. Indexsuche: Auf das Element kann via Index direkt zugegriffen werden.
      ii. Elementsuche: Das Array kann mittels einer Schleife durchlaufen werden.

   In einer Collection hingegen kann das Element nur via Elementensuche gefunden werden.

   (b) Arrays haben keine dynamische Größe, weshalb nur erschwert Elemente eingefügt werden können, wenn das Array schon voll ist. Das kann man bspw. mit der Erstellung eines neuen größeren Arrays und kopieren der Elemente vom Alten ins Neue umgehen. Collections hingegen haben eine variable Größe, d.h. man kann ohne große Bedenken immer neue Elemente einfügen.

   (c) Das Löschen von Elementen in einem Array ist nicht möglich. Sie können bspw. nur durch den Wert `null` überschrieben werden. In einer `java.util .Collection` können Elemente gelöscht werden mittels bspw. `boolean remove(Object o)`.
4. Das Java-Interface `java.util.List` ist nicht mit den Listen aus Racket vergleichbar, da Listen in Racket nicht dynamisch sind. In Java sind Listen von der Größe her deutlich flexibler als in Racket. Ein weiterer Unterschied zwischen `java.util.List` und Listen aus Racket ist, dass die Liste in Racket beliebige Typen enthalten kann. In `java.util.List` können nur Elemente von einem festen Typ gespeichert werden. Der statische Typ bestimmt, welche Typen die Elemente in der Liste haben müssen und der dynamische Typ kann entweder gleich oder eine abgeleitete Klasse vom statischen Typ sein.

Übungsblatt 08: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V2 Collections und Exceptions

Gegeben sei folgender Codeausschnitt:

```
1  double foo(double[] numbers, double n) {
2    LinkedList<Double> list = new LinkedList<>();
3    for (double x : numbers) {
4      if (x > 0 && x <= n && x % 2 != 0) {
5        list.add(x);
6      }
7    }
8    Collections.sort(list);
9    return list.getLast();
10 }
```

(1) Beschreiben Sie kurz und bündig, aber präzise und unmissverständlich was der oben gegebene Code macht.

(2.1) An welcher Stelle kann im Code eine Exception geworfen werden? Durch welche Eingaben wird sie ausgelöst?

(2.2) Modifizieren Sie den Code mithilfe eines `try/catch`-Blockes so, dass in diesen Fällen die Nachricht der Exception auf der Konsole ausgegeben wird.

Lösungsvorschlag:

(1) Die Methode `foo` liefert die größte ungerade Zahl im Array zurück, die kleiner oder gleich `n` und größer als 0 ist.

(2.1)
- Zeile 3: Wenn das übergebene Array gleich `null` ist, so wird eine `NullPointerException` geworfen.
- Zeile 8: Falls das Array keine ungerade Zahl kleiner oder gleich `n` enthält, so wirft der Aufruf von `getLast()` eine `NoSuchElementException`.

(2.2)

```
1  double foo(double[] numbers, double n) {
2    LinkedList<Double> list = new LinkedList<>();
3    try {
4      for (double x : numbers) {
5        if (x > 0 && x <= n && x % 2 != 0) {
6          list.add(x);
7        }
8      }
9    } catch (NullPointerException e) {
10     System.out.println(e.getMessage());
11   }
12   Collections.sort(list);
13   try {
14     return list.getLast();
15   } catch (NoSuchElementException e) {
16     System.out.println(e.getMessage());
17   }
18   return 0;
19 }
```

Übungsblatt 08: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V3 A-well-a bird bird bird, bird is the word Part I

In dieser Aufgabe betrachten wir eine stark reduzierte Typhierarchie zur Modellierung von Vögeln. Dabei stellen die Pfeile die Erbbeziehungen zwischen Klassen dar. Dazu ist das folgende Typdiagramm in Abbildung 1 gegeben. Hier ist `Bird` also die Oberklasse und die drei anderen Klassen sind Erben dieser.

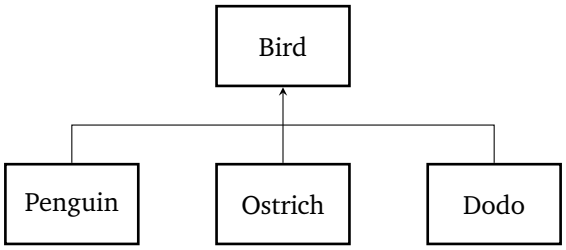


Abbildung 1: Typhierarchie mit drei Vogelarten

Wir nutzen die generische Datenstruktur `Vector<E>`. Dabei beschränken wir uns auf die Methode `void add(E entry)`, die ein Element vom Typ `E` in den Vector einfügt.

(1): Deklarieren und initialisieren Sie eine Variable `v` mit dem **statischen** Basistyp `List` und dem **dynamischen** Typ `Vector`, so dass darin genau Objekte der Typen `Birds`, `Penguin`, `Ostrich` und `Dodo` gespeichert werden können. Nutzen Sie generische Typparameter!

(2): Geben Sie Java-Code an, um in den obigen `Vector v` jeweils ein neues Element vom Typ `Penguin` und `Ostrich` einzufügen. Sie dürfen zur Vereinfachung die Parameter der Konstruktoren der Klassen `Penguin` und `Ostrich` durch ..... abkürzen.

(3): Die Methode `addAll(Birds)` existiert im Interface `List` und fügt eine gesamte `Collection` in eine gegebene `Collection` ein. Die Klasse `ArrayList` implementiert das Interface `List`. Ist die folgende Anweisung – nachdem Code der vorherigen Aufgaben – dann zulässig?

```
1 return new ArrayList<Dodo>().addAll(v);
```

Lösungsvorschlag:

(1):

```
1 List<Bird> v = new Vector<>();
```

(2):

```
1 v.add(new Penguin(...));
2 v.add(new Ostrich(...));
```

(3): Die Anweisung ist nicht zulässig, da `addAll` eine `Collection<? extends Dodo>` erwartet, d.h. nur eine Liste von `Dodo` (Typparameter) oder Klassen, die von `Dodo` abgeleitet sind. Jedoch ist `v` eine Liste von Vögeln (`Bird`).

Übungsblatt 08: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V4 Elemente tauschen

Schreiben Sie eine Methode

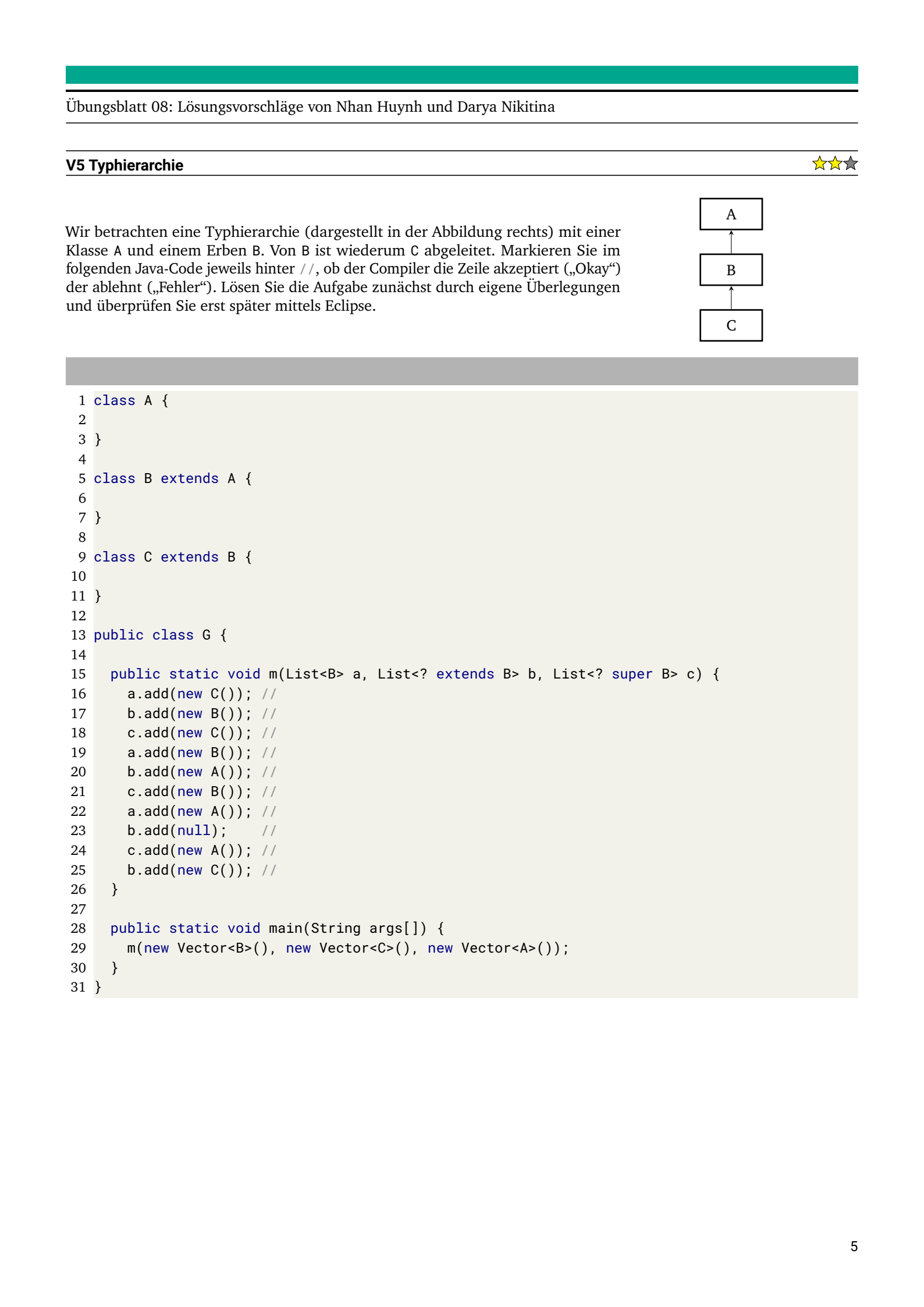
```
void switchElements(T[] a, int i, int j)throws IllegalArgumentException
```

Die Methode vertauscht die Elemente im übergebenen Array `a` an den zwei angegebenen Indizes `i` und `j`. Falls für `a` eine `null`-Referenz übergeben wird oder einer der Indizes nicht in dem Array liegt, soll eine `IllegalArgumentException` geworfen werden.

Lösungsvorschlag:

```
1  /**
2   * Switches the elements at the given indices.
3   *
4   * @param a the array containing the elements
5   * @param i the index of the first element
6   * @param j the index of the second element
7   * @throws IllegalArgumentException if the given array is null or one of indices are out of
8   * bounds
9   */
10 void switchElements(T[] a, int i, int j) throws IllegalArgumentException {
11   if (a == null) {
12     throw new IllegalArgumentException();
13   }
14   int length = a.length;
15   if (i < 0 || j < 0 || i >= length || j >= length) {
16     throw new IllegalArgumentException();
17   }
18
19   // Swap
20   T tmp = a[i];
21   a[i] = a[j];
22   a[j] = tmp;
23 }
```

V5 Typhierarchie



Wir betrachten eine Typhierarchie (dargestellt in der Abbildung rechts) mit einer Klasse `A` und einem Erben `B`. Von `B` ist wiederum `C` abgeleitet. Markieren Sie im folgenden Java-Code jeweils hinter `//`, ob der Compiler die Zeile akzeptiert („Okay“) der ablehnt („Fehler“). Lösen Sie die Aufgabe zunächst durch eigene Überlegungen und überprüfen Sie erst später mittels Eclipse.


```java
class A {

}

class B extends A {

}

class C extends B {

}

public class G {

  public static void m(List<B> a, List<? extends B> b, List<? super B> c) {
    a.add(new C()); //
    b.add(new B()); //
    c.add(new C()); //
    a.add(new B()); //
    b.add(new A()); //
    c.add(new B()); //
    a.add(new A()); //
    b.add(null);    //
    c.add(new A()); //
    b.add(new C()); //
  }

  public static void main(String args[]) {
    m(new Vector<B>(), new Vector<C>(), new Vector<A>());
  }
}
```

Übungsblatt 08: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

Lösungsvorschlag:

```
1  class A {
2
3  }
4
5  class B extends A {
6
7  }
8
9  class C extends B {
10
11 }
12
13 public class G {
14
15   public static void m(List<B> a, List<? extends B> b, List<? super B> c) {
16     a.add(new C()); // Okay
17     b.add(new B()); // Fehler
18     c.add(new C()); // Okay
19     a.add(new B()); // Okay
20     b.add(new A()); // Fehler
21     c.add(new B()); // Okay
22     a.add(new A()); // Fehler
23     b.add(null);    // Okay
24     c.add(new A()); // Fehler
25     b.add(new C()); // Fehler
26   }
27
28   public static void main(String args[]) {
29     m(new Vector<B>(), new Vector<C>(), new Vector<A>());
30   }
31 }
```

Information: Eine Wildcard ist ein spezieller Typparameter für die Instanziierung von generischen (parametrisierten) Typen und wird mit einem `?` gekennzeichnet. Sie wird für die Einschränkung bei dieser Instanziierung verwendet, was eine Flexibilität von Typparametern bei Objekten ermöglicht. Dabei unterscheiden wir zwei Arten von Wildcards. Einmal können die Parameter nach oben durch `extends` beschränkt werden und einmal nach unten durch die Verwendung von `super`:

- Beschränkung der Parameter nach oben
  Bei der Upper bounded Wildcard ist der Typparameter nach oben in der Vererbungshierachie beschränkt. Das ermöglicht es aktuale Parameter zu verwenden, die anstelle von `T` mit einer von `T` abgeleiteten Klassen instanziiert sind.

  Dadurch wird außerdem das Lesen vom Typ `T` (`T` ist ein Platzhalter für einen konkreten Typ) erlaubt, welcher im Wildcard spezifiziert wurde und das Einfügen von `null`. Das Einfügen von anderen Typen ist nicht zulässig, da die Typsicherheit nicht garantiert werden kann. Als Veranschaulichung nehmen wir folgendes Beispiel:

  List<? extends Number>

  Wir wissen, dass der Typparameter der Liste entweder `Number` selbst oder ein Subtyp davon ist, aber nicht genau welcher konkrete Typ es ist. Wir wollen nun einen `Double`-Wert hinzufügen, da `Double` eine Subklasse von `Number` ist. Aber das zu tun wäre nicht typsicher, da der tatsächliche Typparameter der Liste nicht unbedingt `Double` sein muss, sondern auch ein anderer Subtyp oder `Number` selbst sein kann. Beispielsweise könnten, falls die Liste in Wahrheit eine Liste von `Integer` (`ArrayList<Integer>`) ist, keine `Double`-Werte in der Liste gespeichert werden.

- Beschränkung der Parameter nach unten
  Bei einer Lower bounded Wildcard ist der Typparameter nach unten in der Vererbungshierachie beschränkt. Das ermöglicht es anstelle von `T` eine der von `T` direkt oder indirekte Basisklasse, sowie alle von `T` implementierten Interfaces, als aktuale Parameter zu verwenden.

  Dadurch wird außerdem beim Schreiben die Verwendung von `T` selbst oder von dessen Subtyp erlaubt. Das liegt daran, dass `T` oder ein Subtyp von `T` auch indirekt vom Typ deren gemeinsamer Superklasse. Beispielsweise ist `Integer` indirekt auch eine `Number`, da `Integer` ein Subtyp von `Number` ist. Beim Lesen ist nur das Lesen von `Object` garantiert, da es Superklasse von allen Klassen ist. Wir können diese Tatsache an dem folgendes Beispiel veraschaulichen:

  List<? super Number>

  Wir wissen, dass der Typparameter der Liste entweder `Number` oder ein Supertyp davon ist. Das heißt es sind folgende Typen möglich:

  - `Number`
  - `Serializable`
  - `Object`

  In allen Fällen können wir eine `Number` oder einen Subtyp davon einfügen. Wenn wir aber eine `Number` lesen möchten, wäre dies nicht typsicher, da der tatsächliche Typparameter der Liste nicht unbedingt `Number` sein muss, sondern auch ein Supertyp von `Number` sein kann. Deshalb würde beispielsweise, falls die Liste in Wahrheit eine Liste von `Object` (`ArrayList<Object>`) ist, ein Element dessen nicht zwangsweise eine `Number` sein.

V6 A-well-a bird bird bird, bird is the word Part II

Wir erweitern unsere Typhierarchie für Vögel aus Aufgabe nd betrachten neben den nicht-fliegen den Vögeln nun auch ihre flatternden Artgenossen.

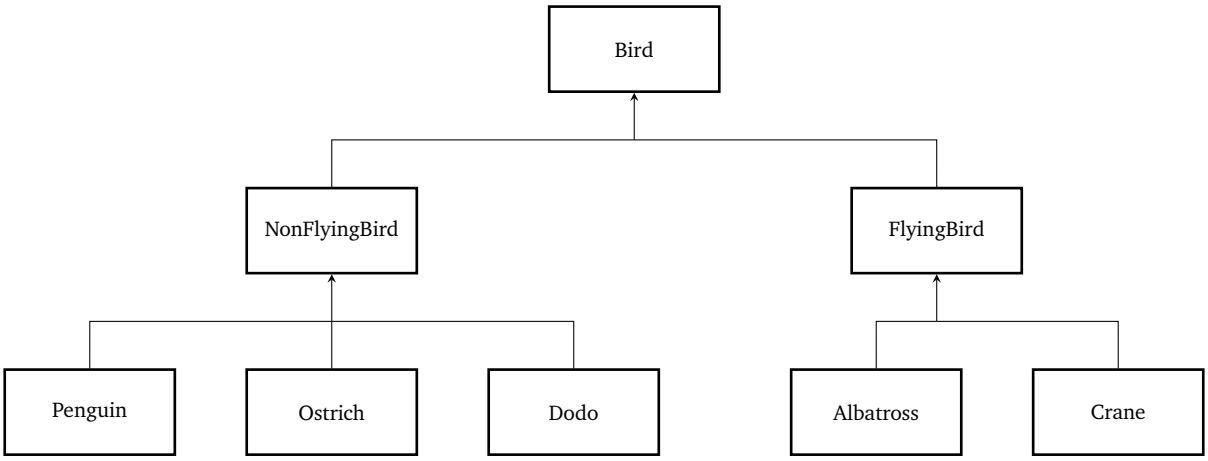


Abbildung 2: Erweiterte Typhierarchie

Vervollständigen Sie die untenstehenden Deklarationen der Methoden. Dabei sind nur die mit ..... markierten Stellen zu bearbeiten! Geben Sie zu jeder Typangabe eine kurze Erklärung, warum genau diese Typangabe die am besten passende oder korrekte ist.
Ihre Lösungen sollen möglichst weitgehend die Typsicherheit garantieren, aber gleichzeitig flexibel für möglichst viele konkrete Parametertypen sein.

Es sind immer generische **Sub**typen zu nutzen.

> **Hinweis:** Zur Vereinfachung wird in den Beispielen nicht auf `null` oder auf eine leere Liste getestet.

(1): Die Methode `getFirst(List<.....> aListOfBirds)` liefert den ersten Vogel einer (nicht-leeren) Liste. Das Ergebnis muss kompatibel zum Typ `Bird` sein.

```
1 Bird getFirst(List<.....> aListOfBirds) {
2   return aListOfBirds.get(0);
3 }
```

(2): Die Methode `void add(Birds b, List<.....> aListOfBirds)` fügt einen neuen Vogel in die Liste ein. Es sollen dabei Vögel jedes bekannten Typs eingefügt werden können.

```
1 void add(Bird b, List<.....> aListOfBirds) {
2   aListOfBirds.add(b);
3 }
```

Übungsblatt 08: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

Lösungsvorschlag:

(1): Laut Aufgabenstellung muss die Liste beliebige Vögel enthalten können und ohne Typecast immer `Birds` liefern. Prinzipiell könnte man hier zwar auch `List<Bird>` nutzen, aber dies schränkt die Menge der möglichen konkreten Typen der als Parameter übergebenen Listen ein und ist daher schlechter. Die Beschränkungen von `List<? extends Birds>` – kein Einfügen von Elementen möglich außer null – sind hier irrelevant.

```
1 Bird getFirst(List<? extends Bird> aListOfBirds) {
2   return aListOfBirds.get(0);
3 }
```

(2): `List<? extends Birds>` und `List<?>` erlauben kein Einfügen von Werten außer `null`. Der Typ `List<Birds>` wäre zu unflexibel, da dabei die Menge der möglichen konkreten Typen der als Parameter übergebenen Listen eingeschränkt wird.

```
1 void add(Bird b, List<? super Bird> aListOfBirds) {
2   aListOfBirds.add(b);
3 }
```

Übungsblatt 08: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V7 Array Utility-Klasse

In dieser Aufgabe wollen wir eine bereits vorhandene Utility-Klasse, für Arrays vom Datentyp `int`, so umschreiben, dass diese für jeden beliebigen Datentyp verwendet werden kann. Die Klasse `ArrayUtils` implementiert folgende Methoden:

`void printArray (int[] array)` bekommt ein `int`-Array übergeben und gibt dessen Elemente auf der Konsole aus.

```
1 public static void printArray(int[] array) {
2   for (int i = 0; i < array.length; i++) {
3     System.out.print(array[i]);
4     if (i < array.length - 1) {
5       System.out.print(" ; ");
6     }
7   }
8   System.out.println();
9 }
```

`int getArrayIndex(int[] array, int value)` bekommt ein `int`-Array und einen Wert übergeben und durchsucht das Array nach dem übergebenen Wert. Wird der Wert gefunden, wird dessen Index im Array zurückgegeben, andernfalls `-1`.

```
1 public static int getArrayIndex(int[] array, int value) {
2   for (int i = 0; i < array.length; i++) {
3     if (array[i] == value) {
4       return i;
5     }
6   }
7   return -1;
8 }
```

`void simpleSort(int[] array)` sortiert das übergebene `int`-Array in aufsteigender Reihenfolge.

```
1 public static void simpleSort(int[] array) {
2   for (int i = 0; i < array.length; i++) {
3     for (int j = i + 1; j < array.length; j++) {
4       if (array[i] > array[j]) {
5         int backup = array[i];
6         array[i] = array[j];
7         array[j] = backup;
8       }
9     }
10  }
11 }
```

Schreiben Sie nun eine Klasse `GenericArrayUtils`, die alle drei oben genannten Methoden mit einem beliebigen Datentyp `T`, bzw. `T[]` für Arrays, implementiert.

> **Hinweis:** Überlegen Sie sich, wie Sie Elemente vom Typ `T` miteinander vergleichen können und welche Voraussetzung dieser Typ `T` mit sich bringen muss. Wie kann sich das im Klassenkopf der zu implementierenden Klasse widerspiegeln?

Lösungsvorschlag:

```
1  /**
2   * A utility class for performing array specific operations.
3   */
4  class ArrayUtils {
5
6    /**
7     * Prints the specified array to the console.
8     *
9     * @param <T>   the type of the elements in the array
10    * @param array the array to be printed
11    */
12   public static <T> void printArray(T[] array) {
13     for (int i = 0; i < array.length; i++) {
14       System.out.print(array[i]);
15       if (i < array.length - 1) {
16         System.out.print(" ; ");
17       }
18     }
19     System.out.println();
20   }
21
22   /**
23    * Returns the index of the specified element in the array, if it exists, else returns
          -1.
24    *
25    * @param <T>   the type of the elements in the array
26    * @param array the array where the element is searched for
27    * @param value the value to be searched in the array
28    * @return the index of the given element in the array if it exists, else return -1.
29    */
30   public static <T> int getArrayIndex(T[] array, T value) {
31     for (int i = 0; i < array.length; i++) {
32       if (array[i] == value) {
33         return i;
34       }
35     }
36     return -1;
37   }
```

```
38
39   /**
40    * Sorts the array.
41    *
42    * @param <T>   the type of the elements in the array
43    * @param array the array to be sorted
44    */
45   public static <T extends Comparable<? super T>> void simpleSort(T[] array) {
46     for (int i = 0; i < array.length; i++) {
47       for (int j = i + 1; j < array.length; j++) {
48         // Swap
49         if (array[i].compareTo(array[j]) > 0) {
50           T backup = array[i];
51           array[i] = array[j];
52           array[j] = backup;
53         }
54       }
55     }
56   }
57 }
```

Übungsblatt 08: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V8 XYZ ★★★

Gegeben seien die folgenden Klassen:

```
1  public class Triple<X, Y, Z> {
2
3    public X x;
4    public Y y;
5    public Z z;
6
7    public Triple(X x, Y y, Z z) {
8      this.x = x;
9      this.y = y;
10     this.z = z;
11   }
12 }
13
14 public class Utils {
15
16 }
```

Erweitern Sie nun die Klasse `Utils` um eine `public`-Klassenmethode `intoMap`, ohne dabei den Klassenkopf zu modifizieren. Die Methode bekommt eine `java.util.List` von Tripeln übergeben und gibt eine `java.util.Map` zurück. Jedes `Triple` in der Liste wird in die `Map` überführt, indem Sie die `x`-Variable des Triples als Schlüssel verwenden und ein neues Paar aus der `y`- und `z`-Variable des Triples erstellen, um dies als Wert des zugehörigen Schlüssels zu verwenden.

Lösungsvorschlag:

```
1  /**
2   * Transforms the list into a map.
3   *
4   * @param input the list to be transformed
5   * @param <X> the type of the key
6   * @param <Y> the type of first element in the pair
7   * @param <Z> the type of the second element in the pair
8   * @return the transformed list as a map
9   */
10 public static <X, Y, Z> Map<X, Pair<Y, Z>> intoMap(List<Triple<X, Y, Z>> input) {
11   Map<X, Pair<Y, Z>> result = new HashMap<>();
12   for (Triple<X, Y, Z> i : input) {
13     result.put(i.x, new Pair<Y, Z>(i.y, i.z));
14   }
15   return result;
16 }
```

Übungsblatt 08: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V9 Matrizenmultiplikation ★★★

In dieser Aufgabe wollen wir eine Matrix Klasse implementieren, die es uns erlaubt jeden beliebigen vergleichbaren Datentyp unter Anwendung von Java Generics mit ihr zu verwenden.

Um Ihnen die Aufgabe zu erleichtern, wird ihnen ein Interface bereitgestellt, welches benutzt werden soll um arithmetische Operationen mit den Matrizen durchzuführen. Bevor man einen Datentyp mit unserer `Matrix` Klasse benutzen kann, muss man zuvor das arithmetische Interface für diesen konkreten Datentyp implementieren. Machen Sie sich mit den Beispielen auf der folgenden Seite vertraut.

Generisches Interface:

```java
/**
 * Defines generic arithmetic operations.
 *
 * @param <T> the type of the arithmetic elements
 */
public interface Arithmetic<T> {

  /**
   * @returns the generic arithmetic representation of zero.
   */
  T zero();

  /**
   * Returns the result of the addition of a and b.
   *
   * @param a the first summand
   * @param b the second summand
   * @return the result of the addition of a and b
   */
  T add(T a, T b);

  /**
   * Returns the result of the multiplication of a and b.
   *
   * @param a the first multiplicand
   * @param b the second multiplicand
   * @return the result of the multiplication of a and b
   */
  T mul(T a, T b);
}
```

Konkrete Implementierung für Gleitkommazahlen mit dem Datentyp Float:

```java
/**
 * Defines float arithmetics.
 */
public class FloatArithmetic implements Arithmetic<Float> {

  @Override
  public Float zero() {
    return 0f;
  }

  @Override
  public Float add(Float a, Float b) {
    return a + b;
  }

  @Override
  public Float mul(Float a, Float b) {
    return a * b;
  }
}
```

Bearbeiten Sie ausgehend davon die Aufgaben auf der nächsten Seite.

V9.1 Erstellen der Matrix-Klasse

Erstellen Sie zunächst die Klasse `public class Matrix<T extends Comparable<T>>`, ie die folgenden aufgezählten Variablen besitzt:

- `private Arithmetic<T> arithmetic` ist zuständig für das Durchführen von arithmetischen Operationen.
- `private LinkedList<LinkedList<T>> data` enthält die Daten der Matrix. Hierbei repräsentiert die äußere `LinkedList` die Reihen und die innere `LinkedList` die Spalten der Matrix.
- `private int rows` wird zum speichern der aktuellen Anzahl der Reihen der Matrix verwendet.
- `private int columns` wird zum speichern der aktuellen Anzahl der Spalten der Matrix verwendet.

Implementieren Sie nun folgende Methoden:

- `public Matrix(int rows, int columns, Arithmetic<T> arithmetic)` erhält die gewünschte Größe der Matrix in Form von Reihen und Spalten und ein entsprechendes Objekt dessen Klasse das arithmetische Interface implementiert. Alle übergebenen Variablen dieser Methode sollen in den entsprechenden Objektvariablen gespeichert werden. Zusätzlich soll jeder Zellenwert mit `arithmetic.zero()` initialisiert werden.
- `public int getRows()` gibt die aktuelle Anzahl der Reihen der Matrix zurück.
- `public int getColumns()` gibt die aktuelle Anzahl der Spalten der Matrix zurück.
- `public T getCell(int row, int column)` erhält den Index einer Reihe sowie den Index einer Spalte und gibt den Wert der Zelle zurück.
- `public void setCell(int row, int column, T value)` erhält den Index einer Reihe sowie den Index einer Spalte und einen gewünschten Wert und setzt diesen an der entsprechenden Stelle in der Matrix ein.

Übungsblatt 08: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

Lösungsvorschlag:

```java
/**
 * Constructs a nxm (rows x columns) matrix.
 *
 * @param rows       the number of rows of the matrix
 * @param columns    the number of columns of the matrix
 * @param arithmetic allows arithmetic operations on the elements in the
 * matrix
 */
public Matrix(int rows, int columns, Arithmetic<T> arithmetic) {
  this.arithmetic = arithmetic;
  this.rows = rows;
  this.columns = columns;
  data = new LinkedList<LinkedList<T>>();
  // Fill matrix with zero
  T zero = arithmetic.zero();
  // Initialize rows by creating a new LinkedList of T
  for (int i = 0; i < rows; i++) {
    data.add(new LinkedList<T>());
    // Initialize each column in the current row with zero
    for (int j = 0; j < columns; j++) {
      data.get(i).add(zero);
    }
  }
}

/**
 * Returns the number of rows of this matrix.
 *
 * @return the number of rows of this matrix
 */
public int getRows() {
  return rows;
}

/**
 * Returns the number of columns of this matrix.
 *
 * @return the number of columns of this matrix
 */
public int getColumns() {
  return columns;
}
```

```
43
44 /**
45  * Returns the element at the specified row and column.
46  *
47  * @param row    the row of the element to be returned
48  * @param column the column of the element to be returned
49  * @return the element at the specified row and column
50  */
51 public T getCell(int row, int column) {
52   return data.get(row).get(column);
53 }
54
55 /**
56  * Puts the value at the specified row and column.
57  *
58  * @param row    the row of the element to be set
59  * @param column the column of the element to be set
60  * @param value  the new value of the cell
61  */
62 public void setCell(int row, int column, T value) {
63   data.get(row).set(column, value);
64 }
```

V9.2 Addition und Multiplikation

Implementieren...

...Sie nun die Methode `public Matrix<T> add(Matrix<T> other)`. Diese erhält eine andere Matrix, addiert die übergebene Matrix und die Matrix, auf der die Methode aufgerufen wurde, miteinander und gibt das Ergebnis der Addition zurück. Sollten die Reihen- und/oder Spaltenanzahl der beiden Matrizen nicht übereinstimmen, soll `null` zurückgegeben werden.

...Sie nun die Methode `public Matrix<T> mul(Matrix<T> other)`. Diese erhält eine andere Matrix, multipliziert die übergebene Matrix und die Matrix, auf der die Methode aufgerufen wurde, miteinander und gibt das Ergebnis der Multiplikation zurück. Sollten die Spaltenanzahl der aktuellen Matrix sich von der Reihenanzahl der übergebenen Matrix unterscheiden, soll `null` zurückgegeben werden.

Übungsblatt 08: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

Lösungsvorschlag:

```java
/**
 * Returns the result of the addition of this matrix with another matrix.
 *
 * @param other the matrix to be added with this matrix
 * @return the result of the addition of this matrix with another matrix
 */
public Matrix<T> add(Matrix<T> other) {
  if (columns != other.columns() || rows != other.rows()) {
    return null;
  }
  Matrix<T> result = new Matrix<T>(rows, columns, arithmetic);
  for (int row = 0; row < rows; row++) {
    for (int column = 0; column < columns; column++) {
      data.get(row).set(column, arithmetic.add(data.get(row).get(column),
          other.getCell(row, row)));
    }
  }
  return result;
}

/**
 * Returns the result of the multiplication of this matrix with another matrix.
 *
 * @param other the matrix to be multiplied with this matrix
 * @return the result of the multiplication of this matrix with another matrix
 */
public Matrix<T> mul(Matrix<T> other) {
  if (columns != other.rows()) {
    return null;
  }
  Matrix<T> result = new Matrix<T>(rows, other.columns, arithmetic);
  for (int row = 0; row < rows; row++) {
    for (int column = 0; column < other.columns; column++) {
      T value = arithmetic.zero();
      for (int k = 0; k < columns; k++) {
        value = arithmetic.add(value, arithmetic.mul(getCell(row, k),
            other.getCell(k, column)));
      }
      result.setCell(row, column, value);
    }
  }
  return result;
}
```