Programmierung 2 (SS 2021)
Universität des Saarlandes
Fakultät MI
Compiler Design Lab

Musterlösung 10
Vererbung, Collections & Hashing

Prof. Dr. Sebastian Hack
Julian Rosemann, B. Sc.
Thorsten Klößner, M. Sc.
Julia Wichlacz, M. Sc.
Maximilian Fickert, M. Sc.

Im Vorlesungskalender finden Sie Informationen über die Kapitel des Skripts, die parallel zur Vorlesung bearbeitet werden sollen bzw. dort besprochen werden. Die Übungsaufgaben dienen der Vertiefung des Wissens, das in der Vorlesung vermittelt wird und als Vorbereitung auf Minitests und Klausur.
Weitere Aufgaben zu den Themen finden Sie jeweils am Ende der Skriptkapitel.
Die Schwierigkeitsgrade sind durch Steine des 2048-Spiels gekennzeichnet, von 512 „leicht“ bis 2048 „schwer“. 4096 steht für Knobelaufgaben.

Vererbung

Aufgabe 10.0 und Aufgabe 10.1 finden Sie ebenfalls auf Blatt 9.

Aufgabe 10.0: Statische und dynamische Typen

Im Folgenden werden Sie schrittweise an die Auflösung von Funktionsaufrufen in Java herangeführt. Die Codeteile sind als zusammenhängend zu interpretieren, die Aufteilung dient der Verdeutlichung der für den Unterpunkt relevanten Programmteile.

1. Bestimmen Sie den statischen und den dynamischen Typ für alle Variablen.

```
1 A a = new A();
2 A b = new B();
3 A c = new C();
4
5 B bB = new B();
6 C cC = new C();
```

2. Wird bei der Überladung der statische oder der dynamische Typ berücksichtigt? Wie sieht es bei der Überschreibung aus?

3. Bestimmen Sie für jede Klasse die Methoden, die überladen sind und diejenigen, die überschrieben werden.

```
1 class A {
2   void f (boolean b) { System.out.println("A.f(boolean)"); }
3
4   void f (double d) { System.out.println("A.f(double)"); }
5 }
6
7 class B extends A {
8   void f (boolean b) { System.out.println("B.f(boolean)"); }
9
10  void f (int i) { System.out.println("B.f(int)"); }
11 }
12
13 class C extends B {
14  void f (boolean b) { System.out.println("C.f(boolean)"); }
15
16  void f (float f) { System.out.println("C.f(float)"); }
17
18  void f (int i) { System.out.println("C.f(int)"); }
19
20  void f (short s) { System.out.println("C.f(short)"); }
21 }
```

4. Bestimmen Sie nun die Ausgabe der folgenden Aufrufe, indem Sie sich für jeden Aufruf die maximal spezifische Signatur überlegen (unter Berücksichtigung des statischen und dynamischen Typen).

```
 1 cC.f(1);
 2 cC.f(1.0f);
 3
 4 a.f(true);
 5 b.f(true);
 6 c.f(true);
 7
 8 bB.f(1);
 9 b.f(1);
10 cC.f(1.0);
11 c.f((short) 1);
```

Lösung

1.

Variable	statischer Typ	dynamischer Typ
a	A	A
b	A	B
c	A	C
bB	B	B
cC	C	C

2. Überladen statisch, Überschreiben dynamisch.

3.

Klasse	Methoden				
A	A.f(double)	A.f(boolean)			
B		B.f(boolean)	B.f(int)		
C		C.f(boolean)	C.f(int)	C.f(short)	C.f(float)

Stehen Methoden in der Tabelle übereinander, so überschreibt die untere die obere Methode. Zum Beispiel `C.f(int)` überschreibt `B.f(int)`. Methoden mit unterschiedlichem Argumenttyp sind überladen. Beispielsweise `B.f(int)` überlädt `A.f(double)`.

4.

```
cC.f(1);          ⟹ C.f(int)
cC.f(1.0f);       ⟹ C.f(float)

a.f(true);        ⟹ A.f(boolean)
b.f(true);        ⟹ B.f(boolean)
c.f(true);        ⟹ C.f(boolean)

bB.f(1);          ⟹ B.f(int)
b.f(1);           ⟹ A.f(double)
cC.f(1.0);        ⟹ A.f(double)
c.f((short) 1);   ⟹ A.f(double)
```

Die maximal spezifische Signatur kommt bei den Aufrufen `b.f(1))` und `c.f((short) 1)` zu tragen, da hier anhand des statischen Typen `A` die Signatur `f(double)` gewählt wird. Da weder in `B` noch in `C` diese Methode definiert wird, wird die Methode aus der Klasse `A` genommen von der `B` und `C` transitiv erben.

Aufgabe 10.1: Überladen und Überschreiben

Betrachten Sie folgenden Code und begründen Sie, welche Methoden in den Zeilen 20–22 aufgerufen werden. Argumentieren Sie mit Überladung und Überschreibung.

```
1 class A {
2   void f (float v) {
3     System.out.println("A.f(float)");
4   }
5 }
6 class B extends A {
7   void f (int v) {
8     System.out.println("B.f(int)");
9   }
10 }
11 class C extends B {
12   void f (int v) {
13     System.out.println("C.f(int)");
14   }
15 }
```

```
16 A a = new B();
17 B b = new B();
18 B c = new C();
19
20 a.f(15);
21 b.f(15);
22 c.f(15);
```

Lösung

Ausgabe:

```
A.f(float)
B.f(int)
C.f(int)
```

`a.f(15)`: a ist eine Objekt der Klasse B, welches nach A gecastet wurde. Da die Signaturen von B.f und A.f nicht identisch sind, wird A.f nicht überschrieben, sondern überladen. Aus diesem Grund ist in der Klasse A die Methode `void f(int v)` (aus Klasse B) nicht mehr sichtbar. 15 wird deshalb zu float gecastet und die Methode A.f wird aufgerufen und `"A.f(float)"` ausgegeben.
`b.f(15)`: 15 ist standardmäßig eine Ganzzahl (int). Dementsprechend wird die überladene Methode `void f (int v)` aufgerufen und `"B.f(int)"` ausgegeben.
`c.f(15)`: In der Klasse C wird mit C.f die Methode B.f überschrieben. Da Methoden in Java dynamisch gebunden werden, wird die Methode C.f aufgerufen, auch wenn c nach B gecastet wurde. Die Ausgabe ist somit `"C.f(int)"`.

Aufgabe 10.2: Vererbung

Betrachten Sie folgende Java-Klassen; sie beschreiben einige Produkte eines Möbelhauses:

```
1 class WoodenTable {
2   private byte   woodspecies;   // 0=unknown, 1=oak, 2=pine, ...
3   private float  price;         // euro
4   private float  weight;        // kg
5   private byte   tshape;        // 0=unknown, 1=rectangular, 2=square, ...
6   private byte   tlegs;         // table legs; >=3
7   private byte   tfunc;         // 0=unkown, 1=dining, 2=desk, 3=drawing
8   // constructor:
9   public WoodenTable(byte wspecies, float price, float weight,
10                     byte tshape, byte   tlegs, byte   tfunc    ) { ... }
11  public String toString() { ... }
12 }
```

```
1 class WoodenChair {
2   private byte   woodspecies;   // 0=unknown, 1=oak, 2=pine, ...
3   private float  price;         // euro
4   private float  weight;        // kg
5   private byte   chshape;       // 0=unknown, 1=chair,  2=stool, 3=armchair
6   private byte   chlegs;        // chair legs; >=3
7   public WoodenChair(byte wspecies, float price, float weight,
8                      byte   chshape, byte   chlegs            ) { ... }
```

```
 9   public String toString() { ... }
10 }
```

```
 1 class WoodenCabinet {
 2   private byte   woodspecies;   // 0=unknown, 1=oak, 2=pine, 3=beech,
 3   private float  price;         // euro
 4   private float  weight;        // kg
 5   private byte   cashape;       // 0=unknown, 1=chair,  2=stool, 3=armchair
 6   private byte   cafunc;        // 0=unknown, 1=, 2=wardrobe, 3=commode, ...
 7   public WoodenCabinet(byte wspecies, float price, float weight,
 8                        byte   cashape, byte cafunc              ) { ... }
 9   public String toString() { ... }
10 }
```

Offensichtlich haben alle 3 Klassen einige Eigenschaften gemeinsam, andere sind für verschiedene Klassen unterschiedlich.

1. Notieren Sie eine neue Klasse namens `WoodenFurniture`, welche die gemeinsamen Eigenschaften aufnimmt. Erstellen Sie hierfür zunächst das Interface `WoodenFurnitureInterface`. Ferner:
   - Passen Sie die gegebenen Klassen `WoodenTable`, `WoodenChair` und `WoodenCabinet` so an, dass Sie von der neuen Klasse `WoodenFurniture` die (gemeinsamen) Eigenschaften nun erben.
   - Passen Sie auch die Konstruktoren an und überlegen Sie sich geeignete `toString`-Methoden.
   - Notieren Sie in einer `main`-Methode geeignete Variablen-Deklarationen und Zuweisungen, um ein Objekt jeder neuen Klasse zu kreieren und die Referenz abzuspeichern.
   - Schreiben Sie eine Methode `sumOfPrices` die als Argument eine Referenz auf eine Reihung von `WoodenFurniture` Referenzen nimmt und die Summe der Preise zurückgibt. Schreiben Sie Getter-Methoden, falls nötig.
2. Ausgehend vom Ergebnis in Teil 1 sollen nun in das Sortiment auch Designer-Holzmöbel aufgenommen werden; sie besitzen als weitere Eigenschaft einen Modellnamen (`String modelname`).
   - Führen Sie neue Klassen namens `DesignerWoodenTable`, `DesignerWoodenChair` und `DesignerWoodenCabinet` mit zugehörigen Konstruktoren ein, indem Sie möglichst viele Anteile per Vererbung wiederverwenden. Die `toString`-Methode dieser Klassen soll nun zusätzlich noch den Modellnamen im Rückgabewert darstellen.

Lösung

1.

```
 1 public interface WoodenFurnitureInterface {
 2   public byte getWoodspecies();
 3
 4   public float getPrice();
 5
 6   public float getWeight();
 7
 8   public String toString();
 9
10 }
```

```
 1 public class WoodenFurniture implements WoodenFurnitureInterface {
 2   private byte woodspecies;
 3   private float price;
 4   private float weight;
 5
 6   public byte getWoodspecies() {
```

```
 7     return woodspecies;
 8   }
 9   public float getPrice() {
10     return price;
11   }
12   public float getWeight() {
13     return weight;
14   }
15   public WoodenFurniture(byte woodspecies, float price, float weight) {
16     this.woodspecies = woodspecies;
17     this.price = price;
18     this.weight = weight;
19   }
20
21   public String toString() {
22     return "Furniture␣of␣wood␣" + woodspecies
23             + "␣with␣weight␣" + weight
24             + "␣for␣only␣" + price;
25   }
26 }
```

```
 1 public class WoodenTable extends WoodenFurniture {
 2   private byte tshape;
 3   private byte tlegs;
 4   private byte tfunc;
 5
 6   public WoodenTable(byte woodspecies, float price, float weight,
 7                      byte tshape, byte tlegs, byte tfunc) {
 8     super(woodspecies, price, weight);
 9     this.tshape = tshape;
10     this.tlegs = tlegs;
11     this.tfunc = tfunc;
12   }
13   public String toString () {
14     return "Table␣with␣shape␣" + tshape + "␣and␣" + tlegs
15            + "␣legs␣for␣activity␣" + tfunc
16            + ",␣" + super.toString();
17   }
18 }
```

```
 1 public class WoodenChair extends WoodenFurniture {
 2   private byte cshape;
 3   private byte chlegs;
 4
 5   public WoodenChair(byte woodspecies, float price, float weight,
 6                      byte cshape, byte chlegs) {
 7     super(woodspecies, price, weight);
 8     this.cshape = cshape;
 9     this.chlegs = chlegs;
10   }
11   public String toString() {
12     return "Chair␣with␣shape␣" + cshape
13            + "␣and␣" + chlegs
14            + "␣legs,␣" + super.toString();
15   }
16 }
```

```
 1 public class WoodenCabinet extends WoodenFurniture {
 2   private byte cashape;
 3   private byte cafunc;
 4
```

```
 5   public WoodenCabinet(byte woodspecies, float price, float weight,
 6                        byte cashape, byte cafunc) {
 7     super(woodspecies, price, weight);
 8     this.cashape = cashape;
 9     this.cafunc = cafunc;
10   }
11   public String toString() {
12     return "Cabinet␣with␣shape␣" + cashape
13            + "␣and␣function␣" + cafunc
14            + ",␣" + super.toString();
15   }
16 }
```

```
 1 public class Main {
 2   public static void main(String[] args) {
 3     WoodenFurniture[] myFurniture = new WoodenFurniture[3];
 4     myFurniture[0] = new WoodenChair((byte)1, 333.0f, 5.0f, (byte)3,
 5                                      (byte)1);
 6     myFurniture[1] = new WoodenTable((byte)2, 4.0f, 4.2f, (byte)1,
 7                                      (byte)4, (byte)2);
 8     myFurniture[2] = new DesignerWoodenCabinet((byte)0, 1000.0f,
 9                                3.1416f, (byte)0, (byte)2,
10                                new DesignerAnnotation("Billy"));
11
12     System.out.println("My␣furniture:");
13     for (WoodenFurniture f : myFurniture) {
14       System.out.println(f);
15     }
16     System.out.println("Price␣for␣all:␣" + sumOfPrices(myFurniture));
17   }
18
19   public static int sumOfPrices (WoodenFurniture[] furniture) {
20     int result = 0;
21     for (WoodenFurniture x : furniture) {
22       result += x.getPrice();
23     }
24     return result;
25   }
26 }
```

2. Wir notieren zunächst eine Klasse `DesignerAnnotation`, die Modellnamen einheitlich behandelt und Erweiterbarkeit der Designeranmerkungen ohne Anpassung der einzelnen Möbel ermöglicht. Die Aufgabenstellung fordert, `String modelname` als Eigenschaft zu verwenden statt dieser Klasse. Diese (triviale) Lösung steht auskommentiert daneben.

```
 1 public class DesignerAnnotation {
 2   private String name;
 3
 4   public DesignerAnnotation(String name) {
 5     this.name = name;
 6   }
 7
 8   public String generateString (String oldstr) {
 9     return "Special␣Design:␣\"" + name + "\",␣" + oldstr;
10   }
11 }
```

```
1 public class DesignerWoodenTable extends WoodenTable {
2   private DesignerAnnotation da;
3   // private String modelname;
4
```

```
 5   public DesignerWoodenTable(byte woodspecies, float price,
 6           float weight, byte tshape, byte tlegs, byte tfunc,
 7           DesignerAnnotation da) {
 8           // String modelname) {
 9     super(woodspecies, price, weight, tshape, tlegs, tfunc);
10     this.da = da;
11     // this.modelname = modelname;
12   }
13
14   public String toString() {
15     return da.generateString(super.toString());
16     // return "Model: "+modelname+", "+super.toString();
17   }
18 }
```

```
 1 public class DesignerWoodenChair extends WoodenChair {
 2   private DesignerAnnotation da;
 3   // private String modelname;
 4
 5   public DesignerWoodenChair(byte woodspecies, float price,
 6           float weight, byte cshape, byte chlegs,
 7           DesignerAnnotation da) {
 8           // String modelname) {
 9     super(woodspecies, price, weight, cshape, chlegs);
10     this.da = da;
11     // this.modelname = modelname;
12   }
13   public String toString() {
14     return da.generateString(super.toString());
15     // return "Model: "+modelname+", "+super.toString();
16   }
17 }
```

```
 1 public class DesignerWoodenCabinet extends WoodenCabinet {
 2   private DesignerAnnotation da;
 3   // private String modelname;
 4
 5   public DesignerWoodenCabinet(byte woodspecies, float price,
 6           float weight, byte cashape, byte cafunc,
 7           DesignerAnnotation da) {
 8           // String modelname) {
 9     super(woodspecies, price, weight, cashape, cafunc);
10     this.da = da;
11     // this.modelname = modelname;
12   }
13   public String toString() {
14     return da.generateString(super.toString());
15     // return "Model: "+modelname+", "+super.toString();
16   }
17 }
```

Aufgabe 10.3: Vererbungshierarchie

Welche Methode wird jeweils aufgerufen? Geben Sie die Ausgabe des Programms an.

Klassenhierarchie

```java
public class A {
 public void foo(A a) {
   System.out.println("A.foo(A)");
 }
 public void foo(B b) {
   System.out.println("A.foo(B)");
 }
}
```

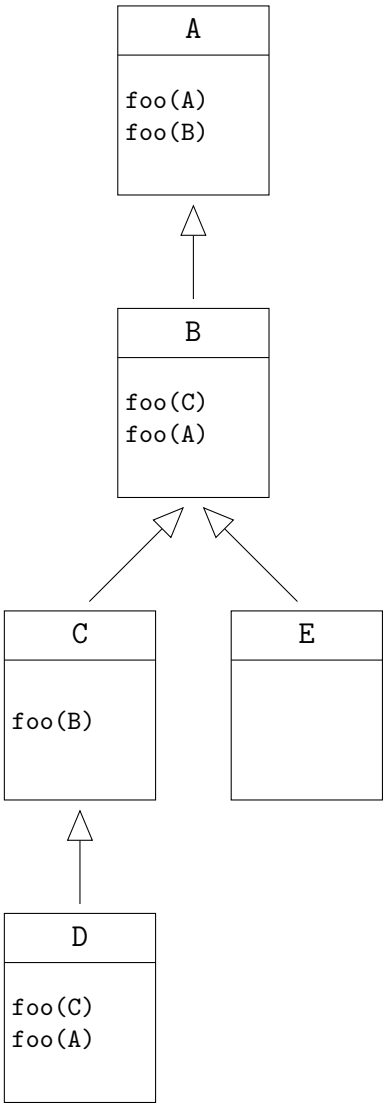
```java
public class B extends A {
 public void foo(A a) {
  System.out.println("B.foo(A)");
 }
 public void foo(C c) {
  System.out.println("B.foo(C)");
 }
}
```

```java
public class C extends B {
 public void foo(B b) {
  System.out.println("C.foo(B)");
 }
}
```

```java
public class D extends C {
 public void foo(A a) {
  System.out.println("D.foo(A)");
 }
 public void foo(C c) {
  System.out.println("D.foo(C)");
 }
}
```

```java
public class E extends B {
}
```

```java
public class Main {
    public static void main(
      String[] args) {
        A aa = new A();
        A ad = new D();
        A ab = new B();
        B bd = new D();
        B be = new E();
        C cd = new D();
        D dd = new D();
        E ee = new E();

        aa.foo(dd);
        ad.foo(be);
        ad.foo(ad);
        ab.foo(bd);

        bd.foo(ee);
        be.foo(dd);

        cd.foo(be);
        cd.foo(ab);

        dd.foo(cd);
        dd.foo(dd);

        ee.foo(be);
        ee.foo(cd);
        ee.foo(ad);

        ad.foo(cd);
    }
}
```



Lösung

```
aa.foo(dd); => A.foo(B)
ad.foo(be); => C.foo(B)
ad.foo(ad); => D.foo(A)
ab.foo(bd); => A.foo(B)

bd.foo(ee); => C.foo(B)
be.foo(dd); => B.foo(C)

cd.foo(be); => C.foo(B)
cd.foo(ab); => D.foo(A)
```

```

dd.foo(cd); => D.foo(C)
dd.foo(dd); => D.foo(C)

ee.foo(be); => A.foo(B)
ee.foo(cd); => B.foo(C)
ee.foo(ad); => B.foo(A)

ad.foo(cd); => C.foo(B)
```

Collections

Aufgabe 10.4: Lotterie

Auf einer Messe will der Veranstalter ein Gewinnspiel ausrichten und legt dazu eine *.txt* Datei an, in welcher er den Namen, die Adresse, die Telefonnummer und eine ganzzahlige Punktzahl (immer in dieser Reihenfolge) speichert. Dateiformat: `Mustername,Musterstrasse. 42 01337 Musterstadt,01234/56789,10`

1. Der Veranstalter merkt, dass bei steigender Teilnehmerzahl die Auswertung schwierig wird und bittet daher Konrad Klug, vermeintlicher Experte für *Collections*, ein Programm zu schreiben. Das Programm soll die *.txt* Datei einlesen und jeden Teilnehmer als Objekt der Klasse *Person* abspeichern.

   Hinweis: Helfen Sie Konrad: Überlegen Sie sich eine passende Collection um die Teilnehmer zu speichern, z.B. wäre *LinkedList* oder *ArrayList* eine erste Möglichkeit.

2. Bestimmen Sie nun die maximale und minimale Punktzahl, sowie den Mittelwert aller Teilnehmer.

3. Sortieren Sie nun die Einträge Ihrer Datenstruktur aufsteigend nach der Punktzahl. Schreiben Sie zusätzlich eine Methode, welche alle Teilnehmer gemäß ihrer Sortierung ausgibt.

4. Als ihr Freund Konrad Klug seiner Freundin No Hau das Programm vorführt ist diese nicht so begeistert, wie er ursprünglich gehofft hat. Sie weist Konrad darauf hin, dass er eine *PriorityQueue* benutzen sollte, um sich das Sortieren zu sparen!

   Hinweis: Es könnte nichts schaden, sich mit *Comparatoren* in Java auseinander zu setzen.

Lösung

(a)

```
1 package root;
2
3 import java.io.BufferedReader;
4 import java.io.FileReader;
5 import java.io.IOException;
6 import java.util.ArrayList;
7
8 public class Main {
9
10   private static ArrayList<Person> participants;
11
12   public static void main(String[] args) {
13     if (args.length == 1) {
14       participants = new ArrayList<Person>();
15
16       try {
17         BufferedReader br = new BufferedReader(new FileReader(args[0]));
18         String line = br.readLine();
19
20         while(line != null) {
21           parseLine(line);
22           line = br.readLine();
23         }
24         br.close();
25       } catch (IOException e) {
26         System.out.println("Datei konnte nicht gelesen werden!");
27       }
28
29       //Stelle 1
30     } else {
31       System.out.println("Bitte geben Sie als Kommandozeilenargument"
```

```
32          + "die␣Datei␣an,␣die␣eingelesen␣werden␣soll!");
33      }
34    }
35
36    public static void parseLine(String line) {
37      String[] person = line.split(",");
38      if (person.length < 4) {
39        System.out.println("Falscher␣Eintrag␣in␣Datei");
40        return;
41      }
42      participants.add(new Person(person[0], person[1], person[2],
43      Integer.parseInt(person[3])));
44    }
45 }
```

```
1 package root;
2
3 public class Person {
4
5   private String name;
6   private String address;
7   private String phonenumber;
8   private int points;
9
10  public Person(String name, String address, String phonenumber,
11      int points) {
12    this.name = name;
13    this.address = address;
14    this.phonenumber = phonenumber;
15    this.points = points;
16  }
17
18  public String getName() {
19    return name;
20  }
21
22  public String getAddress() {
23    return address;
24  }
25
26  public String getPhonenumber() {
27    return phonenumber;
28  }
29
30  public int getPoints() {
31    return points;
32  }
33
34  public String toString() {
35    return this.name + "␣wohnhaft␣in␣" + address +
36    "␣mit␣der␣Telefonnummer␣" + phonenumber + "␣hat␣" + points +
37    "␣erzielt!";
38  }
39 }
```

(b) Füge folgende Methode der Klasse `Main.java` hinzu und rufe diese an der Stelle 1 auf.

```
1 private static void calculateMinMaxAverage() {
2   if(participants.size() > 0) {
3     Person min = participants.get(0);
4     Person max = participants.get(0);
```

```
5     int totalPoints = 0;
6
7     for(Person current : participants) {
8       if(current.getPoints() < min.getPoints()) {
9         min = current;
10      }
11      if(current.getPoints() > max.getPoints()) {
12        max = current;
13      }
14      totalPoints += current.getPoints();
15    }
16
17    double average = (double)totalPoints / participants.size();
18
19    System.out.println("Mittelwert:␣"+ average);
20    System.out.println("Spieler␣mit␣maximaler␣Punktzahl:␣" +
21      max.toString());
22    System.out.println("Spieler␣mit␣minimaler␣Punktzahl:␣" +
23      min.toString());
24  }
25 }
```

(c)

```
1 package root;
2
3 import java.util.Comparator;
4
5 public class PersonComparator implements Comparator<Person> {
6
7   @Override
8   public int compare(Person o1, Person o2) {
9     if(o1.getPoints() < o2.getPoints()) {
10      return -1;
11    }
12    if(o1.getPoints() > o2.getPoints()) {
13      return 1;
14    }
15    return 0;
16  }
17
18 }
```

Füge folgende Methode der Klasse `Main.java` hinzu und rufe diese anstatt der Methode von Aufgabenteil (b) an der Stelle 1 auf.

```
1 private static void calculateRankingCollection() {
2   Collections.sort(participants, new PersonComparator());
3   for(Person current : participants) {
4     System.out.println(current.toString());
5   }
6 }
```

(d)

```
1 package root;
2
3 import java.io.IOException;
4 import java.nio.file.Files;
5 import java.nio.file.Path;
6 import java.nio.file.Paths;
7 import java.util.Comparator;
```

```java
import java.util.PriorityQueue;
import java.util.stream.Stream;

public class Main {

  private static PriorityQueue<Person> participants;
  private static Comparator<Person> compar;

  public static void main(String[] args) {
    if (args.length == 1) {
      compar = new PersonComparator();
      participants = new PriorityQueue<Person>(compar);
      Path input = Paths.get(args[0]);

      try (Stream<String> lines = Files.lines(input)) {
        lines.forEach(s -> parseLine(s));
      } catch (IOException e) {
        System.out.println("Datei konnte nicht gelesen werden!");
      }
      calculateResult();
    } else {
      System.out.println("Bitte geben Sie als Kommandozeilenargument"
        + " die Datei an, die eingelesen werden soll!");
    }
  }

  public static void parseLine(String line) {
    String[] person = line.split(",");
    if (person.length < 4) {
      System.out.println("Falscher Eintrag in Datei");
      return;
    }
    participants.add(new Person(person[0], person[1], person[2],
                                Integer.parseInt(person[3])));
  }

  private static void calculateResult() {
    while(!participants.isEmpty()) {
      System.out.println(participants.poll().toString());
    }
  }
}
```

Aufgabe 10.5: Collections

Java bringt eine große und sehr gut dokumentierte Standardbibliothek mit, die unter anderem die sogenannten *Collections* enthält.

1. Lesen Sie die Erklärung zu den Java-Collections.
2. Überlegen Sie sich zunächst in Eigenarbeit, welche dieser Collections für welche Aufgabe bzw. Problemstellung zur Lösung eingesetzt werden kann.
3. Versuchen Sie sich selbst an der Implementierung einiger der existierenden Collections z. B. `ArrayList` oder `LinkedList`.

Lösung

Wenn Sie hier noch Fragen haben, diskutieren Sie diese im Forum oder schreiben Sie Ihrem Tutor eine Mail.

Aufgabe 10.6: Iterable

Vervollständigen Sie die Klasse `Range<T>`, die ein Intervall $[b, e[$ von ganzen Zahlen darstellt:

```java
public class Range implements Iterable<Integer> {
    private final int begin, end;

    public Range(int begin, int end) {
        assert begin <= end;
        this.begin = begin;
        this.end   = end;
    }

    @Override
    public Iterator<Integer> iterator() {
        // ...
    }
}
```

Implementieren Sie für Ihren Iterator auch die `remove()` Methode. Schlagen Sie vorher nach, was die Defaultimplementierung ist und welche Anforderungen an die Implementieren von `remove()` gestellt werden.

Lösung

```java
import java.util.Iterator;
import java.util.NoSuchElementException;

public class Range implements Iterable<Integer> {
    private final int begin, end;

    public Range(int begin, int end) {
        assert begin <= end;
        this.begin = begin;
        this.end = end;
    }

    @Override
    public Iterator<Integer> iterator() {
```

```
15          return new RangeIterator(begin, end);
16      }
17
18      private static final class RangeIterator implements Iterator<Integer> {
19          private int position;
20          private final int end;
21
22          public RangeIterator(int begin, int end) {
23              this.position = begin;
24              this.end = end;
25          }
26
27          @Override
28          public boolean hasNext() {
29              return this.position < this.end;
30          }
31
32          @Override
33          public Integer next() {
34              if (!this.hasNext()) {
35                  throw new NoSuchElementException();
36              }
37
38              return this.position++;
39          }
40      }
41
42      public static void main(String[] args) {
43          Range range = new Range(1, 10);
44
45          for (Integer current : range) {
46              System.out.println(current);
47          }
48      }
49  }
```

Die Defaultimplementierung von `remove()` wirft lediglich die Ausnahme `UnsupportedOperationException`. Die Methode kann höchstens einmal nach einem Aufruf von `next()` aufgerufen werden, sonst soll die Ausnahme `IllegalStateException` geworfen werden.

```
 1 import java.util.ArrayList;
 2 import java.util.Iterator;
 3 import java.util.NoSuchElementException;
 4
 5 public class Range implements Iterable<Integer> {
 6     private final int begin, end;
 7     private final ArrayList<Integer> removed = new ArrayList<>();
 8
 9     public Range(int begin, int end) {
10         assert begin <= end;
11         this.begin = begin;
12         this.end = end;
13     }
14
15     @Override
16     public Iterator<Integer> iterator() {
17         return new RangeIterator(begin, end, removed);
18     }
19
20     private static final class RangeIterator implements Iterator<Integer> {
21         private int position;
22         private final int end;
```

```
23
24          private boolean removable = false;
25          private final ArrayList<Integer> removed;
26
27          public RangeIterator(int begin, int end,
28                               ArrayList<Integer> removed) {
29              this.position = begin;
30              this.end = end;
31              this.removed = removed;
32          }
33
34          @Override
35          public boolean hasNext() {
36              if (this.position >= this.end)
37                  return false;
38              int removedIndex = removed.indexOf(position);
39              if (removedIndex == -1)
40                  return true;
41              return removed.size() - removedIndex != end - position;
42          }
43
44          @Override
45          public Integer next() {
46              if (!this.hasNext()) {
47                  throw new NoSuchElementException();
48              }
49              removable = true;
50              while (removed.contains(this.position))
51                  this.position++;
52              return this.position++;
53          }
54
55        @Override
56          public void remove() {
57              if (!this.removable) {
58                  throw new IllegalStateException();
59              }
60              removable = false;
61              removed.add(position - 1);
62              removed.sort(Integer::compareTo);
63        }
64      }
65
66      public static void main(String[] args) {
67          Range range = new Range(1, 10);
68
69          Iterator<Integer> iterator = range.iterator();
70          Integer curr;
71          while (iterator.hasNext()) {
72              curr = iterator.next();
73              if (curr % 2 == 0)
74                  iterator.remove();
75          }
76
77          for (Integer current : range) {
78              System.out.println(current);
79          }
80      }
81  }
```

Aufgabe 10.7: Farmer John und seine Kühe

Konrad Klug wurde von seinem Freund Farmer John angerufen, der seine Hilfe braucht. Nachdem seine Lieblingskuh Bessie das Stromkabel angebissen hat, spielt die Kuh-Datenbank (eine Liste aller Kühe) verrückt und eine Kuh scheint doppelt vorzukommen. Konrad Klug hat bereits begonnen die Aufgabe zu lösen, kommt jedoch nicht weiter. Vervollständigen Sie seine Implementierung und helfen Sie Farmer John.

1. Konrad Klugs Idee ist es die Liste zu sortieren und dann die sortierte Liste nach mehrfachen Auftreten zu durchsuchen. Implementieren Sie dazu folgende Methode und verwenden Sie Collections.sort(). Überlegen Sie sich, wie Sie die Kuh-Klasse verändern müssen, um Collections.sort() korrekt verwenden zu können.

```
1 public Kuh doppelteKuh(List<Kuh> kuehe){}
```

2. No Hau hat gehört, dass es effizienter ist ein HashSet zu verwenden und beim Einfügen zu überprüfen, ob sich das Element bereits im Set befindet. Implementieren Sie dazu folgende Methode indem Sie ein Hashset verwenden:

```
1 public Kuh effizienteDoppelteKuh(List<Kuh> kuehe){}
```

```
1 public class Kuh {
2     private String name = "";
3     private int fleckenAnzahl = 0;
4
5     public Kuh(String name, int fleckenAnzahl) {
6         assert name != null;
7         this.name = name;
8         this.fleckenAnzahl = fleckenAnzahl;
9     }
10
11 }
```

Lösung

```
1 public class Kuh implements Comparable<Kuh>{
2     private String name = "";
3     private int fleckenAnzahl = 0;
4
5     public Kuh(String name, int fleckenAnzahl) {
6         assert name != null;
7         this.name = name;
8         this.fleckenAnzahl = fleckenAnzahl;
9     }
10
11     @Override
12     public boolean equals(Object object) {
13         if (this == object)
14             return true;
15
16         if (!(object instanceof Kuh))
17             return false;
18
19         Kuh kuh = (Kuh) object;
20         return (this.name.equals(kuh.name)
21             && this.fleckenAnzahl == kuh.fleckenAnzahl );
22     }
23
24     @Override
25     public int hashCode() {
26         return 31 * name.hashCode() + fleckenAnzahl;
```

```
27      }
28
29      @Override
30      public int compareTo(Kuh kuh) {
31          if (fleckenAnzahl == kuh.fleckenAnzahl){
32              return name.compareTo(kuh.name);
33          }
34
35          return fleckenAnzahl - kuh.fleckenAnzahl;
36      }
37  }
```

1.

```
1 public Kuh doppelteKuh(List<Kuh> kuehe){
2     Collections.sort(kuehe);
3     for (int i = 1; i < kuehe.size(); i++){
4         if (kuehe.get(i).equals(kuehe.get(i-1))){
5             return kuehe.get(i);
6         }
7     }
8     return null;
9 }
```

2.

```
1 public Kuh effizienteDoppelteKuh(List<Kuh> kuehe){
2     Set<Kuh> set = new HashSet<>();
3     for (Kuh kuh : kuehe){
4         if (set.contains(kuh)){
5             return kuh;
6         }
7         set.add(kuh);
8     }
9     return null;
10 }
```

Hashing

Aufgabe 10.8: hashCode und equals

Im Folgenden sind zwei Klassen `Fraction` und `Str` gegeben, die jeweils Brüche und Zeichenketten darstellen sollen. Implementieren Sie für beide sinnvolle `hashCode()` und `equals()` Methoden, die die entsprechenden Java-Konventionen einhalten.

a)

```java
public class Fraction {

    private final int numerator, denominator;

    public Fraction(int numerator, int denominator) {
        if (denominator == 0) throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public int getNumerator() {
        return numerator;
    }

    public int getDenominator() {
        return denominator;
    }
}
```

b)

```java
public class Str {

    private final byte[] values;

    public Str(byte... values) {
        this.values = values;
    }

    @Override
    public String toString() {
        return String.valueOf(values);
    }
}
```

Lösung

a)

```java
public class Fraction {

    private final int numerator, denominator;

    public Fraction(int numerator, int denominator) {
        if (denominator == 0)
            throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }

```

```
12      public int getNumerator() {
13          return numerator;
14      }
15
16      public int getDenominator() {
17          return denominator;
18      }
19
20      @Override
21      public int hashCode() {
22          // make numerator and denominator unique by computing their
23          // greatest common divisor and dividing by it
24          int gcd = computeGcd(numerator, denominator);
25          return numerator / gcd + (denominator / gcd) * 31;
26
27          // Alternative way to do it:
28          // It may suffer errors introduced by floating point arithmetic
29          //return new Double((double) numerator / denominator).hashCode();
30      }
31
32      private static int computeGcd(int a, int b) {
33          return b == 0 ? a : computeGcd(b, a % b);
34      }
35
36      @Override
37      public boolean equals(Object obj) {
38          if (!(obj instanceof Fraction))
39              return false;
40          Fraction other = (Fraction) obj;
41          // this ensures that 3 / 4 and 6 / 8 are equal
42          return this.numerator * other.denominator ==
43           this.denominator * other.numerator;
44      }
45  }
```

b)

```java
public final class Str {

    private final byte[] values;

    public Str(byte... values) {
        this.values = values;
    }

    @Override
    public int hashCode() {
        final int prime = 29;
        int hash = 1;
        // uses the Horner schema
        for (byte b : values) {
            hash = prime * hash + b;
        }
        return hash;
    }

    @Overrided
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Str other = (Str) obj;
        if (values.length != other.values.length)
            return false;
        for (int i = 0; i < values.length; i++) {
            if (values[i] != other.values[i])
                return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return String.valueOf(values);
    }
}
```

Die Klasse `String` in `java.lang.String` ist `final`. Das bedeutet, von dieser Klasse kann nicht geerbt werden. Diese Eigenschaft wurde hier übernommen. Deshalb wird in der Methode `equals` aus Effizienz `getClass` an Stelle von `instanceof` verwendet - letzteres ist jedoch nicht falsch. Bedenken Sie, dass bei der Verwendung von `getClass` separat auf `null` getestet werden muss.

Aufgabe 10.9: Kollisionslisten

Konrad Klug möchte seine Lieblingsinformatiker in einem `HashSet` speichern. Dazu hat er folgende Klasse angelegt und eine Hashfunktion implementiert.
`quersumme(int x)` berechnet dabei die Quersumme von `x`.


```
1    public class Wissenschaftler {
2      private String name;
3      private short tag, monat, jahr;
4
5      public int hashCode() {
6        return quersumme(tag) + quersumme(monat) + quersumme(jahr);
7      }
8    }
```

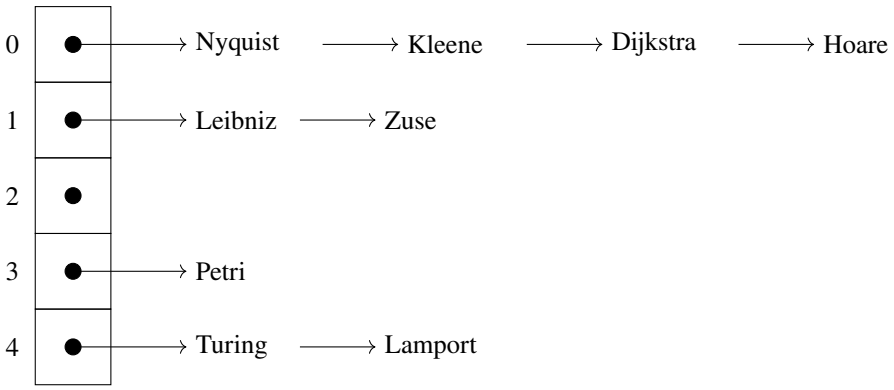
Name	Geburtstag
Alan Turing	23.6.1912
Gottfried Wilhelm Leibniz	21.6.1646
Harry Nyquist	07.2.1889
Leslie Lamport	07.2.1941
Stephen Cole Kleene	05.1.1909
Edsger Wybe Dijkstra	11.5.1930
Konrad Ernst Otto Zuse	22.6.1910
Carl Adam Petri	12.7.1926
Charles Antony Richard Hoare	11.1.1934

Tabelle 1: Wissenschaftler

Hashen Sie die Wissenschaftler aus Tabelle 1 in ein HashSet der Größe 5 ein. Verwenden Sie zur Kollisionsauflösung Kollisionslisten.

Lösung

Name	Geburtstag	Quersumme	Platz
Alan Turing	23.6.1912	24	4
Gottfried Wilhelm Leibniz	21.6.1646	26	1
Harry Nyquist	07.2.1889	35	0
Leslie Lamport	07.2.1941	24	4
Stephen Cole Kleene	05.1.1909	25	0
Edsger Wybe Dijkstra	11.5.1930	20	0
Konrad Ernst Otto Zuse	22.6.1910	21	1
Carl Adam Petri	12.7.1926	28	3
Charles Antony Richard Hoare	11.1.1934	20	0



Aufgabe 10.10: Lineares Sondieren

Konrad Klug hat seine Meinung geändert. Er möchte die Kollisionen nun doch mittels linearem Sondieren auflösen.

1. Hashen Sie die Wissenschaftler aus Tabelle 1 in ein HashSet der Größe 9 ein. Verwenden Sie zur Kollisionsauflösung lineares Sondieren.
2. Löschen Sie Leibniz wieder aus dem HashSet. Danach möchte Konrad überprüfen, ob Nyquist in seinem HashSet enthalten ist. Welches Problem tritt auf und wie können Sie es beheben?

Lösung

1.

0	Nyquist
1	Kleene
2	Dijkstra
3	Zuse
4	Petri
5	Hoare
6	Turing
7	Lamport
8	Leibniz

Name	Geburtstag	Quersumme	Platz
Alan Turing	23.6.1912	24	6
Gottfried Wilhelm Leibniz	21.6.1646	26	8
Harry Nyquist	07.2.1889	35	8
Leslie Lamport	07.2.1941	24	6
Stephen Cole Kleene	05.1.1909	25	7
Edsger Wybe Dijkstra	11.5.1930	20	2
Konrad Ernst Otto Zuse	22.6.1910	21	3
Carl Adam Petri	12.7.1926	28	1
Charles Antony Richard Hoare	11.1.1934	20	2

2. Wenn nur getestet wird, ob an Position 8 ein Element enthalten ist, wird Nyquist nicht gefunden. Es muss markiert werden, ob an der entsprechenden Position mal ein Element enthalten war um gegebenenfalls weiter zu suchen.

Aufgabe 10.11: Quadratisches Sondieren

Konrad Klug hat seine Meinung nochmal geändert. Er möchte die Kollisionen nun doch mittels quadratischem Sondieren auflösen.

1. Hashen Sie die Wissenschaftler aus Tabelle 1 in folgender Reihenfolge in ein HashSet der festen Größe 9 ein:
   `Turing Kleene Lamport Nyquist Dijkstra Leibniz Petri Zuse Hoare`
   Verwenden Sie zur Kollisionsauflösung quadratisches Sondieren mit der Funktion:

$$h'(x,i) = \left(h(x) + \frac{i(i+1)}{2}\right) \; mod \; 9$$

2. Was fällt Ihnen auf?

Lösung

1.

0	Lamport
1	Petri
2	Dijkstra
3	Zuse
4	
5	Leibniz
6	Turing
7	Kleene
8	Nyquist

2. Hoare kann nicht im freien 4. Platz eingefügt werden, da für kein i gilt: `h'(Hoare, i) = 4`

Aufgabe 10.12: Cuckoo Hashing

Konrad Klug ist begeistert von Hashing und möchte neben linearem und quadratischen Sondieren weitere praxisrelevante Sondierungsmethoden finden. Bei seinen Recherchen stößt Konrad auf **Cuckoo** und **Robin-Hood Hashing**, welche er versucht auf selbst generierte Beispiele anzuwenden. Jedoch muss er feststellen, dass er die Theorie der beiden Methoden noch nicht ganz verstanden hat. Helfen Sie ihm dabei! (Lesen Sie sich hierbei die Theorie hinter Cuckoo und Robin Hood Hashing im Internet an).

Gegeben sei folgende Hashtabelle:

0	1	2	3	4	5	6

1. Gegeben seien folgende Hashfunktionen für die benötigten Hashtabellen: $h_1(x) = (x+1)^2 - 3$, $h_2(x) = 2 * x$. Fügen sie 2,7,11,1,8,15 in die Hashtabelle mit Cuckoo Hashing ein.

2. Gegeben sei folgende Hashfunktion: $h_1(x) = 2x - 1$. Fügen sie 2,7,11,1,8,15 in die Hashtabelle mit Robin-Hood-Hashing ein.

Lösung

1.

0	1	2	3	4	5	6
	8			7		2

0	1	2	3	4	5	6
	11	1				

Will man nun die 15 einhashen, entsteht wieder eine Kollision und man versucht die 8 in das zweite Array einzuhashen. Hier kommt es erneut zu einer Kollision, wodurch die 8 in das zweite Array eingesetzt und respektive die 1 in das erste Array eingesetzt wird. Da wir nun die 15 aus dem ersten Array herausnehmen müssen und in das zweite Array einsetzen, versuchen wir die 8 in das erste Array einzuhashen. Nun muss die 1 wieder in das zweite und verdrängt somit die 15. Jetzt haben wir einen Zyklus gefunden, da wir die Zahl die wir ursprünglich einhashen wollten, erneut in das erste Array einfügen müssten. Wir verdoppeln nun die Größe des Arrays. Anschließend müssen alle Werte neu eingehasht werden:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	15				7	2		8					

0	1	2	3	4	5	6	7	8	9	10	11	12	13
		1						11					

2. Nach dem Einhashen der Werte und dem Lösen der Kollisionen sieht die Hashtabelle wie folgt aus:

0	1	2	3	4	5	6
11	1	8	15	2		7
0	1	1	1	3		6

Aufgabe 10.13: Veränderter Inhalt

Betrachten Sie folgendes Programm:

```java
import java.util.HashSet;

public class Main {

    public static void main(String[] args) {
        HashSet<Lorex> uhren = new HashSet<Lorex>();
        Lorex r1 = new Lorex("Explorer", 5600, 500);
        Lorex r2 = new Lorex("Pre-Daytona", 12050, 700);

        uhren.add(r1);
        uhren.add(r2);
        System.out.println(uhren.contains(r1));
        System.out.println(uhren.contains(r2));

        r1.setPreis(4643);
        r2.setPreis(19040);
        System.out.println(uhren.contains(r1));
        System.out.println(uhren.contains(r2));
    }
```

```
20 }
```

```
1 public class Lorex {
2  String name;
3  int preis;
4  int goldanteil;
5
6  public Lorex(String name,
7     int preis, int goldanteil) {
8   this.goldanteil = goldanteil;
9   this.preis = preis;
10  this.name = name;
11 }
12
13 public void setPreis(int preis) {
14  this.preis = preis;
15 }
```

```
16 @Override
17 public boolean equals(Object obj) {
18  if (this == obj)
19    return true;
20  if (obj == null)
21    return false;
22  if (getClass() != obj.getClass())
23    return false;
24  Lorex other = (Lorex) obj;
25  if (!name.equals(other.name))
26    return false;
27  return true;
28 }
29
30 @Override
31 public int hashCode() {
32  return (goldanteil + preis) % 10;
33 }
34 }
```

1. Was ist die Ausgabe der `main`-Methode?
2. Überrascht Sie diese Ausgabe? Wobei liegt hier das Problem?

Lösung

1.
```
System.out.println(uhren.contains(r1));  =>  true
System.out.println(uhren.contains(r2));  =>  true

System.out.println(uhren.contains(r1));  =>  false
System.out.println(uhren.contains(r2));  =>  true
```

2. Wir wissen, dass 2 Objekte, die sich im Sinne von `equals` gleichen, den gleichen Hashwert besitzen müssen. Das ist hier jedoch nicht der Fall. Mit dem Überschreiben von `equals` definieren wir 2 Objekte der Klasse `Lorex` als gleich, wenn sie denselben Namen besitzen. Unsere Hashfunktion bezieht sich jedoch nur auf die Felder `goldanteil` und `preis`, welche nichts mit `equals` zu tun haben. Deshalb wird in Zeile 17 der `main`-Methode irrtümlicherweise `false` ausgegeben, obwohl sich lediglich der Preis von `r1` und nicht ihr Befinden in `uhren` verändert hat. Eine bessere Hashfunktion würde sich demnach auf `name` beziehen und sieht wie folgt aus:

```
1  @Override
2  public int hashCode() {
3    return name.hashCode();
4    }
```