Aufgabe 1: Vererbung und Abstrakte Klassen

In einer Anforderungsanalyse für ein Banksystem wird der folgende Sachverhalt beschrieben:

Eine Bank hat einen Namen und sie führt Konten. Jedes Konto hat eine Kontonummer, einen Kontostand und einen Besitzer. Der Besitzer hat einen Namen und eine Kundennummer. Ein Konto ist entweder ein Sparkonto oder ein Girokonto. Ein Sparkonto hat einen Zinssatz, ein Girokonto hat einen Kreditrahmen und eine Jahresgebühr.

(a) Deklarieren Sie geeignete Klassen in Java [oder in C++], die die oben beschriebenen Anforderungen widerspiegeln! Nutzen Sie dabei das Vererbungskonzept aus, wo es sinnvoll ist! Gibt es Klassen, die als abstrakt zu verstehen sind?

[Hinweis: Geben Sie sowohl ein Klassendiagramm als auch den Quellcode für die beteiligten Klassen inkl. Attributen, ohne Methoden an! Achtung: in Teilaufgabe b) werden anschließend die benötigten Konstruktoren verlangt; Um die verschiedenen Konten in einer Bank zu verwalten, eignet sich das Array NICHT als Datenstruktur. Informieren Sie sich über die Datenstruktur „ArrayList“ und verwenden Sie diese.]

> Die Klasse Konto ist abstrakt, da ein Konto immer entweder ein Spar- *oder* ein Girokonto ist. Ein Objekt der Klasse Konto ist deshalb nicht sinnvoll.

(b) Geben Sie für alle nicht abstrakten Klassen benutzerdefinierte Konstruktoren an mit Parametern zur Initialisierung der folgenden Werte: der Name einer Bank, die Kontonummer, der Kontostand, der Besitzer und der Zinssatz (bzw. Kreditrahmen und Jahresgebühr) eines Sparkontos (bzw. Girokontos), der Name und die Kundennummer eines Kontobesitzers.

Ergänzen Sie die Klassen um Methoden für die folgenden Aufgaben! Nutzen Sie wann immer möglich das Vererbungskonzept aus und verwenden Sie ggf. abstrakte (bzw. virtuelle) Methoden!

[Achtung: Ergänzen Sie ggf. alle benötigten Getter und Setter in den beteiligten Klassen!]

> Konstruktoren ergänzen:
>
> Bemerkung: Auch eine abstrakte Klasse kann einen Konstruktor besitzen, dieser kann nur nicht ausgeführt werden. In den abgeleiteten Klassen kann dieser Super-Konstruktor aber verwendet werden.

(c) Auf ein Konto soll ein Betrag eingezahlt und ein Betrag abgehoben werden können. Soll von einem Sparkonto ein Betrag abgehoben werden, dann darf der Kontostand nicht negativ werden. Bei einer Abhebung von einem Girokonto darf der Kreditrahmen nicht überzogen werden.

> Bemerkung: Die Methode `einzahlen` ist in der Klasse `Konto` implementiert, da sie sich für Spar- und Girokonten nicht unterscheidet im Gegensatz zur Methode `abheben`, die in beiden Klassen unterschiedlich implementiert ist. [Sie wird als abstrakte Methode in der Klasse Konto angegeben, um ihre Implementierung (Überschreiben) zu gewährleisten.] Kreditrahmen wird als negativer Wert gespeichert. Die Methoden zum Abheben liefern

zusätzlich zur Änderung des Kontostandes eine Rückmeldung bezüglich Erfolg oder Misserfolg der Abbuchung.

(d) Ein Konto kann eine Jahresabrechnung durchführen. Bei der Jahresabrechnung eines Sparkontos wird der Zinsertrag gutgeschrieben, bei der Jahresabrechnung eines Girokontos wird die Jahresgebühr abgezogen (auch wenn dadurch der Kreditrahmen überzogen wird).

Anmerkung: Im Folgenden nur noch Angabe der gesuchten Methoden, alle vorherigen Implementierungen (Attribute, Konstruktoren, Methoden, s. o.) wurden nicht nochmals aufgeführt.

(e) Eine Bank kann einen Jahresabschluss durchführen. Dieser bewirkt, dass für jedes Konto der Bank eine Jahresabrechnung durchgeführt wird.

(f) Eine Bank kann ein Sparkonto eröffnen. Die Methode soll die folgenden fünf Parameter haben: den Namen und die Kundennummer des Kontobesitzers, die Kontonummer, den (anfänglichen) Kontostand und den Zinssatz des Sparkontos. Alle Parameter sind als String-Objekte oder als Werte eines Grunddatentyps zu übergeben! Das Sparkonto muss nach seiner Eröffnung in den Kontenbestand der Bank aufgenommen sein.

[Hinweis: Falls der Kunde schon mit einem Konto in der Bank geführt ist, können die Werte für das `Besitzer`-Objekt übernommen werden. Schreiben Sie daher eine Hilfsmethode `Besitzer schonVorhanden(String name, int kunr)`, die prüft, ob der Kunde mit `name` und `kunr` schon in der Liste vorhanden ist und diesen bzw. andernfalls einen neu erzeugten Besitzer zurückgibt.]

[Sinnvolle/notwendige Methoden der Klasse ArrayList für diese Aufgabe:
`public int size()`: Returns the number of elements in this list.
`public boolean isEmpty()`: Returns true if this list contains no elements.
`public E get(int index)`: Returns the element at the specified position in this list.
`public boolean add(E e)`: Appends the specified element to the end of this list.
(siehe auch Java-Api: https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html)]

```
3  import java.util.ArrayList;
4
5  public class Bank {
6    @SuppressWarnings("unused")
7    private String name;
8    private ArrayList<Konto> fuehrt;
9
10   public Bank(String name) {
11     this.name = name;
12     fuehrt = new ArrayList<Konto>();
13   }
14
15   public void abrechnung() {
16     for (int i = 0; i <= fuehrt.size(); i++) {
17       fuehrt.get(i).abrechnung(); // Methode " get ( i ) " liefert i .
         ↪ Listenelement zurueck
18     }
19   }
20
```

```
21    public void anlegen(String n, int kn, int kto, float kst, float zs)
      ↪  {
22      Besitzer bes = schonVorhanden(n, kn);
23      fuehrt.add(new Sparkonto(kto, kst, bes, zs)); // Methode " add (
        ↪  element ) " haengt neues Element ans Ende der Liste
24    }
25
26    public Besitzer schonVorhanden(String name, int kunr) {
27      if (!fuehrt.isEmpty()) {
28        for (int i = 0; i < fuehrt.size(); i++) {
29          if (fuehrt.get(i).hatBesitzer.getKundennr() == kunr) {
30            // prueft , ob die Kundennr . des Kontos an der i . Stelle ,
              ↪  des zugehoerigen
31            // Kontobesitzers " hatBesitzer " gleich kunr ist
32            return fuehrt.get(i).getHatBesitzer();
33          }
34        }
35      }
36      return new Besitzer(name, kunr);
37
38    }
39
40  }
```

```
3   public abstract class Konto {
4     protected int kontonr;
5     protected float kontostand;
6     protected Besitzer hatBesitzer;
7
8     public Konto(int knr, float kst, Besitzer b) {
9       kontonr = knr;
10      kontostand = kst;
11      hatBesitzer = b;
12
13    }
14
15    public void einzahlen(float betrag) {
16      kontostand += betrag;
17    }
18
19    public Besitzer getHatBesitzer() {
20      return hatBesitzer;
21    }
22
23    public abstract boolean abheben(float betrag);
24
25    public abstract void abrechnung();
26
27  }
```

```
3   public class Sparkonto extends Konto {
4     private float zinssatz;
5
6     public Sparkonto(int knr, float kst, Besitzer b, float zs) {
7       super(knr, kst, b);
8       zinssatz = zs;
9     }
10
11    public boolean abheben(float betrag) {
12      if (kontostand - betrag >= 0) {
13        kontostand -= betrag;
```

```
      return true;
    } else {
      return false;
    }
  }

  public void abrechnung() {
    kontostand += kontostand * (1 + zinssatz);
  }
}
```

```
public class Girokonto extends Konto {
  private float kreditrahmen;
  private float jahresgebuehr;

  public Girokonto(int knr, float kst, Besitzer b, float kr, float jg)
  ↪ {
    super(knr, kst, b);
    kreditrahmen = kr;
    jahresgebuehr = jg;
  }

  public boolean abheben(float betrag) {
    if (kontostand - betrag >= kreditrahmen) {
      kontostand -= betrag;
      return true;
    } else {
      return false;
    }
  }

  public void abrechnung() {
    kontostand -= jahresgebuehr;
  }
}
```

```
public class Besitzer {
  @SuppressWarnings("unused")
  private String name;
  private int kundennr;
  @SuppressWarnings("unused")
  private Konto hatKonto;

  public Besitzer(String name, int kundennr) {
    this.name = name;
    this.kundennr = kundennr;
  }

  public int getKundennr() {
    return kundennr;
  }
}
```