1 Grundlagen

In diesem Projekt wirst du verschiedene Möglichkeiten kennen lernen, Objekte in Sammlungen zusammenzufassen.

Beispiele sind:

a) Bibliotheken verwalten Informationen über Bücher und Zeitschriften.
b) Schulsekretariate verwalten Informationen über Schüler.
c) Lehrer verwalten Informationen über Noten einer Klasse.
d) Universitäten verwalten Informationen über ehemalige und momentane Studenten.

Ein typisches Problem ist, dass sich die Anzahl der zu verwaltenden Elemente in einer Sammlung ständig verändert. Zur Zeit des Zwischenzeugnisses treten einige Schüler in eine andere Klasse freiwillig zurück. In einer Bibliothek werden neue Bücher angeschafft und alte werden archiviert bzw. entsorgt.

Du wirst nun in Java *Datenstrukturen* kennen lernen, die die Gruppierung einer beliebigen Anzahl von Elementen erlaubt. Im Laufe dieses Projekts erstellst du eine einfache Anwendung für ein Telefonbuch.

1.1 Benutzung von Sammlungen

1.1.1 Sammlungen vom Typ List

Die Abbildung 1.1. zeigt den Quelltext des Projekts *TelefonList*.

```
import java.util.*;

public class Telefonbuch
{
    private ArrayList<String> buch;

    public Telefonbuch()
    {
        buch = new ArrayList<String>();
    }

    /** Fuegt einen neuen Personennamen in das Telefonbuch ein. */
    public void neuerName(String name)
    {
        buch.add(name);
    }

    /** Zeigt einen bestimmten Personennamen. */
```

```
    public String gibName(int index)
    {
        String name = buch.get(index);
        return name;
    }

    /** Liefert die Anzahl der gespeicherten Namen. */
    public int gibAnzahlNamen()
    {
        int anzahl = buch.size();
        return anzahl;
    }
}
```

Abbildung 1.1: Quelltext zu *TelefonList*

Die Klasse *ArrayList* ist im Paket *java.util* enthalten. Anschließend wird das Datenfeld *buch* vom Typ *ArrayList* definiert. In dieser *List* kannst du alle Namen sammeln. Im Konstruktor wird nun ein Objekt der Klasse *ArrayList* erzeugt und im Datenfeld *buch* gespeichert.

Die Klasse *ArrayList* definiert nun einige Methoden; im folgenden Beispiel verwendest du von diesen die Methoden *add(), get()* und *size()*.

Übung 1.1:

Erstelle in **BlueJ** das Projekt *TelefonList* und implementiere die Klasse *Telefonbuch*. Untersuche nun die Methoden *neuerName(), gibName()* und *gibAnzahlNamen().*

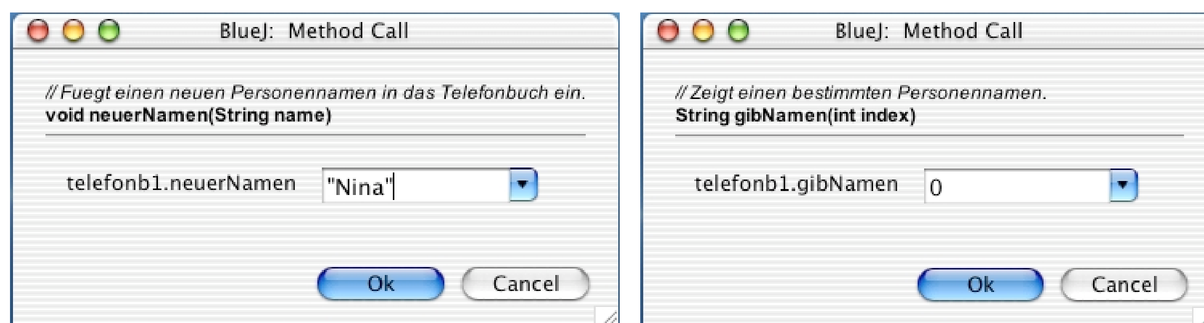

Abbildung 1.2: Die Methoden *neuerName()* und *gibName()*

Übung 1.2:

Verwende den Inspektor und überprüfe folgende Behauptungen:

1. Die Klasse *ArrayList* kann ihre interne Kapazität bei Bedarf vergrößern. Werden weitere Objekte eingefügt, wird für diese einfach Platz

geschaffen.

2. Die Klasse *ArrayList* merkt sich die Anzahl der aktuell gespeicherten Elemente.

3. Die Klasse *ArrayList* behält die Reihenfolge bei, in der die Elemente eingefügt wurden.

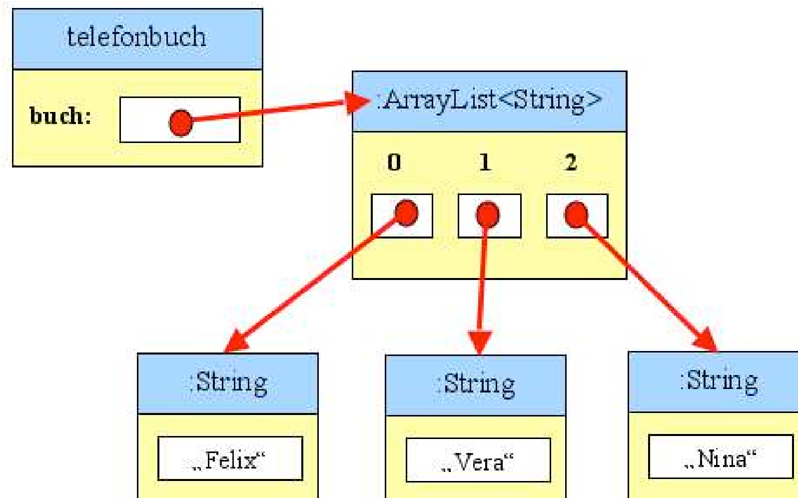

Abbildung 1.3: Modell des Telefonbuchs

Die Abbildung 1.3 illustriert ein Modell des Telefonbuchs. Alle Elemente in einer Sammlung haben eine bestimmte Position, beginnend mit dem Index 0. Weitere Elemente werden mit der Methode *add()* am Ende angefügt. Die Methode *get()* zeigt dasjenige Element an der übergebenen Position *index*.

Der Typ des Datenfelds *buch* wurde deklariert als *ArrayList<String>*. Die Klasse heißt *ArrayList* und erfordert die Angabe eines zweiten Typs als Parameter. Klassen, die einen solchen Parameter erfordern, heißen *generische Klassen*. Die Klasse *ArrayList* kann also benutzt werden, um eine *ArrayList von Strings*(*ArrayList<String>*), eine *ArrayList von Personen* (*ArrayList<Person>*), eine *ArrayList von Konten* (*ArrayList<Konto>*), usw. zu definieren.

private ArrayList<String> name;
private ArrayList<Person> adressbuch
private ArrayList<Konto> bank

Das Objekt *name* hält eine ArrayList, in der nur String-Objekte gespeichert werden, während *adressbuch* eine *ArrayList* halten kann, die *Person*-Objekt speichert, usw.

Übung 1.3:

Simuliere die in Abbildung 1.3 dargestellte Situation und rufe die Methode *gibName()* mit dem Parameter 3 auf.

Ein häufiger Fehler ist, dass bei einem Zugriff auf ein Element einer Sammlung der Index außerhalb der gültigen Indexgrenzen einer *ArrayList* liegen. Die Abbildung 1.3 zeigt nur drei Elemente, die Methode *gibName(3)* versucht jedoch auf ein viertes Element zuzugreifen. Es wurde also eine Index-Grenzverletzung verursacht (engl.: *index-out-of-bounds*).

Übung 1.4:

Implementiere die Methode *gibName(),* so dass bei einer Index-Grenzverletzung eine Fehlermeldung erscheint.

Genauso einfach ist es, Elemente aus einer Sammlung zu entfernen. Die Klasse *ArrayList* stellt hierzu die Methode *remove()* zur Verfügung. Dieser Methode wird der Index des zu entfernenden Elements übergeben.

```
/** Entfernt einen bestimmten Personennamen aus dem Telefonbuch. */
public void entferneName(int index)
{
    buch.remove(index);
}
```

Abbildung 1.4: Die Methode *entferneName()*

Übung 1.5:

Implementiere die Methode *entferneName()* aus Abbildung 1.4.
Verwende den Inspektor und überprüfe die Nummerierung der Elemente, wenn ein Element aus der Mitte der Sammlung entfernt wird.
Überprüfe das Entfernen eines Elements außerhalb der Indexgrenzen. Ergänze die Methode mit geeigneten Fehlermeldungen.

1.1.2 Durchlaufen einer Sammlung

Das Einfügen und Entfernen von Namen bedeutet, dass die Indexnummern der jeweiligen Elemente in einer Sammlung sich ständig ändern können. Interessant wäre nun eine Methode, die jedes Element einer Sammlung auflistet.

Eine Möglichkeit wären Anweisungen nach folgendem Muster:

```
System.out.println(buch.get(0));
System.out.println(buch.get(1));
System.out.println(buch.get(2));
………
```

Leider hängt die Anzahl der Anweisungen ab von der Anzahl der Namen, die zu einem bestimmten Zeitpunkt im Telefonbuch eingetragen sind. Mit Hilfe von Schleifen lässt sich dieses Problem lösen. Schleifen können verwendet werden, um einen Block von Anweisungen wiederholt ausführen zu lassen, ohne dass diese Anweisungen mehrfach im Quelltext geschrieben werden müssen.

Ich werde dir nun drei Möglichkeiten vorstellen, das oben genannte Problem zu lösen.

1.1.2.1 Die for-each-Schleife

Die Abbildung 1.5 zeigt eine Implementierung der Methode *gibAlleNamen()*, die alle Namen mit ihrem zugehörigen Index in der Sammlung im Terminal-Fenster ausgibt.

```
/** Listet alle Namen auf (for-each-Schleife). */
public void zeigeAlleNamen()
{
    for (String name : buch) {
        int index = buch.indexOf(name);
        System.out.println(index + " : " + name);
    }
}
```

Abbildung 1.5: *for-each*-Schleife

Vor jeder Ausführung der beiden zu wiederholenden Anweisungen im Rumpf der Schleife wird der lokalen Variablen *name* vom Typ *String* eines der Elemente in der Liste zugewiesen, das mit dem Index 0 als erstes, mit Index 1 als zweites, usw. Im Schleifenrumpf kannst du dann anschließend dich mit Hilfe dieser Schleifenvariablen auf dieses Element beziehen. So wird nacheinander jedes Element in der Liste *buch* abgearbeitet.

Übung 1.6:

Implementiere die Methode *zeigeAlleNamen()* aus Abbildung 1.5. Erzeuge ein Telefonbuch und speichere darin einige Namen. Überprüfe, ob die Methode *zeigeAlleNamen()* auch dann noch korrekt funktioniert, wenn nachträglich weitere Namen eingefügt bzw. entfernt wurden.

Verwende den Debugger, um zu beobachten, wie die Anweisungen in der Schleife abgearbeitet.

1.1.2.2 Die while-Schleife

Übung 1.7:

Informiere Dich über das Konzept von Schleifen. Vergleiche dazu im Skript „Einführung in BlueJ, 11. Jahrgangsstufe“ das Kapitel „Algorithmen“.

Übung 1.8:

Implementiere mit Hilfe einer *while*-Schleife die Methode *zeigeAlleNamen()*, die alle Namen im Telefonbuch zusammen mit deren Index auflistet. Erzeuge ein Telefonbuch und speichere darin einige Namen. Überprüfe, ob die Methode *zeigeAlleNamen()* auch dann noch korrekt funktioniert, wenn nachträglich weitere Namen eingefügt bzw. entfernt wurden.
Verwende den Debugger, um zu beobachten, wie die Anweisungen in der Schleife abgearbeitet.

1.1.2.3 Der Iterator

Neben der Verwendung einer *while*-Schleife existiert eine weitere Möglichkeit, eine Sammlung zu durchlaufen. Die Klasse *ArrayList* liefert mit Hilfe der Methode *iterator()* ein *Iterator*-Objekt. Ein Iterator bietet die beiden Methoden *hasNext()* und *next()*, mit denen über eine Sammlung iteriert werden kann.

```
/** Listet alle Namen auf (Iterator). */
public void zeigeAlleNamen()
{
    Iterator<String> it = buch.iterator();
    while (it.hasNext()) {
        String name = it.next();
        System.out.println(name);
    }
}
```

Abbildung 1.6: Verwendung eines Iterators in der Methode *zeigeAlleNamen()*

Übung 1.9:

Implementiere die Methode *zeigeAlleNamen()* aus Abbildung 1.6.Erzeuge ein Telefonbuch und speichere darin einige Namen. Überprüfe, ob die Methode

zeigeAlleNamen() auch dann noch korrekt funktioniert, wenn nachträglich weitere Namen eingefügt bzw. entfernt wurden.

Verändere die Methode so, dass zusätzlich die Indizes der jeweiligen Namen angezeigt werden.

Bemerkung:

Bei Verwendung von Strings lässt sich allerdings die Methode *zeigeAlleNamen()* kürzer schreiben:

```
public void zeigeAlleNamen()
{
    Iterator<String> it = buch.iterator();
    while(it.hasNext()) {
        System.out.println(it.next());
    }
}
```

Alle drei Versionen funktionieren gleich gut. Vielleicht fällt dir der erste Ansatz mit der *while*-Schleife einfacher oder dieser ist leichter zu verstehen. Bei einer Sammlung vom Typ *ArrayList* sind alle drei Möglichkeiten in der Tat gleich gut. Jedoch bietet Java neben der *ArrayList* noch eine weitere Anzahl von Sammlungsklassen. Für einige dieser Sammlungen ist es unmöglich, über einen Index auf die einzelnen Elemente zuzugreifen. Die Lösung mit Hilfe eines Iterators wird hingegen von allen Sammlungen unterstützt.

Übung 1.10:

Füge in dein Telefonbuch einen gleichen Namen zweimal ein und überprüfe die Einträge mit Hilfe der Methode *zeigeAlleNamen()*. Überprüfe auch die Einträge mit Hilfe des Inspektors. Erkläre die auftretenden Probleme!

Du hast in diesen Übungen einige wichtige Eigenschaften der Sammlung *List* kennen gelernt. Eine *List* hält ihre Elemente in der Reihenfolge, in der sie eingeben werden und kann dasselbe Element mehrfach enthalten. Zum Durchlaufen einer List bietet sie den Zugriff über einen Index oder über einen Iterator oder sie bietet die Verwendung der *for-each*-Schleife.

1.1.3 Sammlungen vom Typ Set

Ein *Set* ähnelt einer *List*, erlaubt jedoch im Gegensatz zu dieser keine doppelten Elemente. Wenn ein Element ein zweites Mal eingefügt wird, hat dies keinen Effekt. Ein *Set* definiert außerdem für ihre Elemente keine Reihenfolge. Das

bedeutet, der Iterator kann Elemente in einer anderen Reihenfolge liefern als die, in der sie eingefügt wurden. Somit macht auch der Zugriff über einen Index keinen Sinn mehr.

```
import java.util.*;

public class Telefonbuch
{
    private HashSet<String> buch;

    public Telefonbuch()
    {
        buch = new HashSet<String>();
    }

    /** Fuegt einen neuen Personennamen in das Telefonbuch ein. */
    public void neuerName(String name)
    {
        buch.add(name);
    }

    /** Listet alle Namen auf (for-each-Schleife). */
    public void zeigeAlleNamen()
    {
        for (String name : buch) {
            System.out.println(name);
        }
    }
}
```

Abbildung 1.7: Quelltext zu *TelefonSet1*

Übung 1.11:

Erstelle in **BlueJ** das Projekt *TelefonSet1* und implementiere die Klasse *Telefonbuch* aus Abbildung 1.7.

Füge die drei Personen **Nina**, **Vera** und **Felix** ein und überprüfe die Einträge mit Hilfe der Methode *zeigeAlleNamen()*. Anschließend füge zusätzlich die beiden Namen **Nina** und **Peter** ein und überprüfe die Einträge wiederum. Was stellst Du fest?

Übung 1.12:

Implementiere eine Methode *zeigeAlleNamen()*, die mit Hilfe eines Iterators alle im *buch* gespeicherten Namen auflistet.

Eine weitere interessante Eigenschaft ist die Realisierung von sortierten Mengen.

Übung 1.13:

Speichere das Projekt *TelefonSet1* unter dem Namen *TelefonSet2*. Ersetze in den beiden Anweisungen das Schlüsselwort *HashSet* durch *TreeSet*.

Füge nun beliebige Namen in das Telefonbuch ein und liste diese auf. Füge anschließend weitere Namen ein und liste diese Erweiterung auf.

Formuliere schriftlich, was du festgestellt hast!

1.1.4 Sammlungen vom Typ Map (für Experten)

Ein Telefonbuch wird in der Regel als Nachschlagewerk verwendet. Man sucht nach einem bestimmten Namen und erhält die zugehörige Telefonnummer:

Schlüssel	→	**Wert**
Nina	→	111
Vera	→	222
Felix	→	333

Im Prinzip ist also ein Telefonbuch eine Abbildung, die jedem Namen eine Telefonnummer zuordnet, anders ausgedrückt, ein Schlüssel wird auf einen Wert abgebildet. Solche Abbildungen werden mit Hilfe der Klasse *Map* realisiert. Eine *Map* enthält also Schlüssel-Werte-Paare.

Auf die einzelnen Elemente einer *Map* kannst du nicht mit Hilfe eines Indizes zugreifen. In einem Telefonbuch suchst du auch nicht die Position eines Eintrags, sondern du verwendest einen Namen und suchst die zugeordnete Telefonnummer. Abbildungen sind somit ideal für eine einseitige Suche, wenn du den Suchschlüssel kennst und den zugehörigen Wert benötigst.

```
import java.util.*;

public class Telefonbuch
{
    private HashMap<String, String> buch;

    public Telefonbuch()
    {
        buch = new HashMap<String, String>();
    }
```

```
    /** Fuegt einen neuen Nameen mit Nummer in das Telefonbuch ein. */
    public void neuerName(String name, String nummer)
    {
        buch.put(name, nummer);
    }

    /** Zeigt einen bestimmten Personennamen. */
    public String gibNummer(String name)
    {
        String nummer = (String) buch.get(name);
        return nummer;
    }
}
```

Abbildung 1.8: Quelltext zu *TelefonMap1*

Übung 1.14:

Erstelle in **BlueJ** das Projekt *TelefonMap1* und implementiere die Klasse *Telefonbuch* aus Abbildung 1.8. Beantworte anschließend folgende Fragen:

a) Was passiert, wenn du einen Eintrag mit einem Schlüssel vornimmst, der bereits eingetragen ist?

b) Was passiert, wenn du einen Eintrag mit einem Wert vornimmst, der bereits eingetragen ist?

c) Was passiert, wenn du einen Wert suchst und der Schlüssel in der *Map* nicht existiert?

d) Implementiere eine Methode *gibAnzahl(),* die die Anzahl der Einträge zurück gibt?

Das Auflisten aller Einträge in einer *Map* gestaltet sich schwierig. Die Klasse *Map* besitzt keine Methode *iterator(),* um von einem Schlüssel-Wert-Paar zum nächsten zu gelangen. Stattdessen kennt sie drei Methoden:

keySet() — Sie liefert die Menge der Schlüssel. Da keine doppelten Schlüssel in einer *Map* auftauchen, ist diese Sammlung vom Typ *Set*.

values() — Sie liefert die Menge der Werte. Da Werte doppelt vorkommen dürfen, ist der Rückgabewert lediglich vom Typ *Collection*.

entrySet() — Sie liefert die Menge von Schlüssel-Werte-Paare. Jedes Element dieser Menge ist vom Typ *Map.Entry*. Dieses stellt u.a. die Methoden *getKey()* und *getValue()* zur Verfügung, um auf die beiden Komponenten des Paares zurückzugreifen.

Mit Hilfe eines Iterators kannst du über all diese drei Mengen iterieren.

```
public void zeigeAlleNamen()
{
    for (String name : buch.keySet()) {
        System.out.println(name);
    }
}
```

Abbildung 1.9: Verwendung der Methode *keySet()* liefert die Schlüssel

```
public void zeigeAlleNummern()
{
    for (String nummer : buch.values()) {
        System.out.println(nummer);
    }
}
```

Abbildung 1.10: Verwendung der Methode *values( )* liefert die Werte

```
public void zeigeAlleEintraege1()
{
    for (Map.Entry entry : buch.entrySet()) {
        String name = (String) entry.getKey();
        String nummer = (String) entry.getValue();
        System.out.println(name + " --> " + nummer);
    }
}
```

Abbildung 1.11: Verwendung der Methode *entrySet()* mit *getKey()* und *getValue()*

Eine Alternative zur Verwendung von *entrySet*() zeigt die Methode *zeigeAlleEintraege2().*

```
public void zeigeAlleEintraege2()
{
    Iterator it = buch.keySet().iterator();
    while (it.hasNext()) {
        String name = (String) it.next();
        String nummer = (String) buch.get(name);
        System.out.println(name + " --> " + nummer);
    }
}
```

Abbildung 1.12: Alternative zu *entrySet()*

Übung 1.15:

Implementiere im Projekt *TelefonMap1* die vier obigen Methoden und untersuche deren Wirkungsweise.

Versuche die obigen Methoden auch mit Hilfe eines Iterators bzw. mit Hilfe einer *for-each*-Schleife zu implementieren.

Eine weitere interessante Eigenschaft ist die Realisierung von sortierten Mengen.

Übung 1.16:

Speichere das Projekt *TelefonMap1* unter dem Namen *TelefonMap2*. Ersetze in den beiden Anweisungen das Schlüsselwort *HashMap* durch *TreeMap*.

Füge nun beliebige Schlüssel-Werte-Paare in das Telefonbuch ein und liste diese auf. Füge anschließend weitere Paare ein und liste diese Erweiterung auf.

Formuliere schriftlich, was du festgestellt hast!

1.1.5 Zusammenfassung

Die Klasse *Collection* (Sammlung) wird im Wesentlichen in den drei Grundformen *List, Set* und *Map* realisiert.

Eine *List* ist eine beliebig große Liste von Elementen beliebigen Typs, auf die sowohl wahlfrei als auch sequentiell zugegriffen werden kann.

Ein *Set* ist eine duplikatenfreie Menge von Elementen, auf die mit Mengenoperationen zugegriffen werden kann.

Eine *Map* ist eine Abbildung von Elementen eines Typs auf Elemente eines anderen Typs, also eine Menge zusammengehöriger Paare von Objekten.

<table>
<tr><th colspan="7">Implementations</th></tr>
<tr><th rowspan="4">Interfaces</th><th></th><th>Hash Table</th><th>Resizable Array</th><th>Balance Tree</th><th>Linked List</th></tr>
<tr><td>Set</td><td>HashSet</td><td></td><td>TreeSet</td><td></td></tr>
<tr><td>List</td><td></td><td>ArrayList</td><td></td><td>LinkedList</td></tr>
<tr><td>Map</td><td>HashMap</td><td></td><td>TreeMap</td><td></td></tr>
</table>

Abbildung 1.13: Entnommen aus „The Java Tutorial Continued, The Rest of the JDK“, S.68

Jede dieser Grundformen ist als Interface unter dem links angegebenen Namen implementiert. Zudem gibt es jeweils eine oder mehrere konkrete Implementierungen.

Eine *Collection* wird in der Regel mit Hilfe einer *for-each*-Schleife oder eines Iterators durchlaufen. Bei einer *List*-Collection gibt der Iterator die Elemente in der Reihenfolge ihrer Indexnummern zurück. Bei der *HashSet*- bzw. *HashMap*-Collection gibt der Iterator die Elemente in keiner bestimmten Reihenfolge zurück. Bei der *TreeSet*- bzw. *TreeMap*-Collection werden die Elemente sortiert.

1.2 Alphabetische Ordnung im Telefonbuch

1.2.1 Die Klasse Person

Eine Person ist gekennzeichnet durch seinen Nachnamen, seinen Vornamen und seine Telefonnummer. Diese Eigenschaften werden in den Datenfeldern *nachname*, *vorname* und *telefon* gespeichert und im Konstruktor mit den entsprechenden Werten belegt.

Abbildung 1.14: Modell der Klasse *Person*

Weiterhin besitzt eine Person die Fähigkeiten *gibNachname()*, *gibVorname()*, *gibTelefon()*, mit deren Hilfe diese Person ihre Daten zurück liefern kann.

```
public class Person
{
    private String nachname, vorname, telefon;

    public Person(String nachnameH, String vornameH, String telefonH)
    {
```

```
        nachname = nachnameH;
        vorname = vornameH;
        telefon = telefonH;
    }

    public String gibNachname()
    {
        return nachname;
    }

    public String gibVorname()
    {
        return vorname;
    }

    public String gibTelefon()
    {
        return telefon;
    }
}
```

Abbildung 1.15: Die Klasse *Person* im Projekt *Telefon1a*

1.2.2 Die Klasse Telefonbuch vom Typ *Set*

Die Klasse *Telefonbuch* verwaltet nun ihre Einträge in einer Sammlung vom Typ *HashSet*. Somit vermeidest du doppelte Einträge, was auch sinnvoll ist. Jedoch werden in der Sammlung *buch* nun keine Strings mehr verwaltet, sondern Objekte der Klasse *Person*.

```
import java.util.*;
public class Telefonbuch
{
    private HashSet<Person> buch;

    public Telefonbuch()
    {
        buch = new HashSet<Person>();
    }

    public void neuePerson(Person person)
    {
        if (person != null) {
            buch.add(person);
        }
    }
```

```
    public void zeigeAllePersonen()
    {
        for (Person person : buch) {
            String nachname = person.gibNachname();
            String vorname = person.gibVorname();
            String telefon = person.gibTelefon();
            System.out.println(nachname + " " + vorname + " " + telefon);
        }
        System.out.println();
    }
}
```

Abbildung 1.16: Die Klasse *Telefon* im Projekt *Telefon1a*

Übung 1.17:

Erstelle in **BlueJ** das Projekt *Telefon1a* und implementiere die Klassen *Person* und *Telefonbuch* aus den Abbildungen 1.15 und 1.16. Untersuche anschließend folgende Probleme:

a) Erzeuge folgende drei Objekte der Klasse *Person* (Straka Tobias 111), (Schurian Alexander 222), (Heider Teresa 333) und ein Objekt der Klasse *Telefonbuch*. Füge mit Hilfe der Methode *neuePerson()* die drei Personen in das Telefonbuch ein und rufe die Methode *zeigeAllePersonen()* auf. In welcher Reihenfolge werden die Daten dieser Personen dargestellt? Begründe dies schriftlich!

b) Erzeuge zwei weitere Personen (Riedel Franziska 444), (Schurian Isabella 555) und füge diese zusätzlich in das Telefonbuch ein. In welcher Reihenfolge werden nun die Daten dieser Personen dargestellt

Die Abbildung 1.17 zeigt, dass die Personen nicht in alphabetischer Reihenfolge aufgelistet werden. Die Reihenfolge der Personen in deinem Telefonbuch kann auch von der Reihenfolge in Abbildung 1.17 abweichen. Da du jedoch eine Sammlung vom Typ *HashSet* verwendet hast, ist dies auch verständlich.

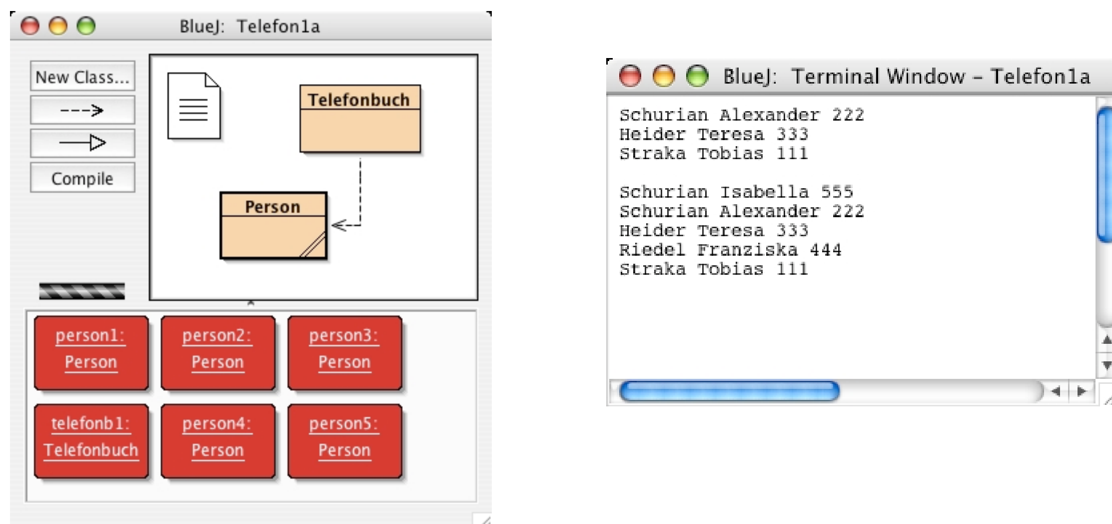

Abbildung 1.17: Die Personen werden nicht sortiert in der Sammlung *HashSet*

Um nun die Personen in alphabetischer Reihenfolge aufzulisten, wird in der Klasse *Telefonbuch* das Schlüsselwort *HashSet* durch *TreeSet* ersetzt.

Übung 1.18

Speichere das Projekt *Telefon1a* unter dem Namen *Telfon1b*. Ersetze in der Klasse *Telefonbuch* das Schlüsselwort *HashSet* durch *TreeSet*.

Erzeuge anschließend zwei Objekte der Klasse *Person* und ein Objekt der Klasse *Telefonbuch*. Füge mit Hilfe der Methode *neuePerson()* diese beiden Personen in das Telefonbuch ein und rufe die Methode *zeigeAllePersonen()* auf.

Welches Problem taucht nun auf?

Bei der *TreeSet*-Collection werden die Elemente nun alphabetisch sortiert zurückgeliefert. Dazu muss jedes Element der Sammlung die Methode *compareTo()* besitzen, damit zwei beliebige Elemente miteinander verglichen werden können.

Bemerkung:
Seit dem JDK 1.2 wird das *Comparable*-Interface bereits von vielen der eingebauten Klassen implementiert, etwa von *String*, *Character*, *Double*, usw. Die natürliche Ordnung einer Menge von Elementen ergibt sich nun, indem alle Elemente paarweise miteinander verglichen und dabei jeweils das kleinere vor das größere Element gestellt wird.
(aus „Handbuch der Java-Programmierung“ von G. Krüger)

Da du bisher in den Übungen 1.13 bzw. 1.16 nur Strings in den Sammlungen verwaltet hast, brauchtest du dir also um die natürliche Ordnung keine Gedanken machen, weil in der Klasse *String* das *Comparable*-Interface bereits implementiert ist. Werden jedoch in diesen Sammlungen beliebige Objekte z.B. Objekte einer Klasse *Person* verwaltet, musst du mit der Methode *compareTo*() eine Ordnung selbst erzeugen.

compareTo()	liefert einen Wert kleiner 0, wenn das aktuelle Element vor dem zu vergleichenden liegt,
	liefert einen Wert größer 0, wenn das aktuelle Element hinter dem zu vergleichenden liegt,
	liefert den Wert 0, wenn das aktuelle Element und das zu vergleichenden gleich sind.

An Hand des zurückgegebenen Werts *vergleich* kann nun die Ordnung erstellt werden.

```
public class Person implements Comparable

……….. weitere Anweiszungen ………..

public int compareTo(Object jene)
{
    Person jenePerson = (Person) jene;
    int vergleich = nachname.compareTo(jenePerson.gibNachname());
    if (vergleich != 0) {
        return vergleich;
    }
    vergleich = vorname.compareTo(jenePerson.gibVorname());
    if (vergleich != 0) {
        return vergleich;
    }
    vergleich = telefon.compareTo(jenePerson.gibTelefon());
    return vergleich;
}
```

Abbildung 1.18: Die Methode *compareTo()* in der Klasse *Person*

Bemerkung:

Eine genauere Einführung in das *Comparable*-Interface würde hier zu weit führen. Der interessierte Leser kann jedoch mit Hilfe von *System.out.println()*-Anweisungen und mit Hilfe des Inspektors erforschen, welchen Wert die Variable *vergleich* annimmt und wie ein Objekt *Person* seine Position innerhalb

in der *TreeSet*-Sammlung *buch* findet, wenn es neu in das Telefonbuch eingefügt wird.

Übung 1.19:

Implementiere in dem Projekt *Telefon1b* nun die Methode *compareTo()* in der Klasse *Person*.

Erzeuge folgende fünf Objekte der Klasse *Person* (Straka Tobias 111), (Schurian Alexander 222), (Heider Teresa 333), (Riedel Franziska 444), (Schurian Isabella 555) und ein Objekt der Klasse *Telefonbuch*. Füge mit Hilfe der Methode *neuePerson()* die fünf Personen in das Telefonbuch ein und rufe die Methode *zeigeAllePersonen()* auf.

Das Terminal sollte folgenden Eintrag zeigen (in alphabetischer Sortierung):

Heider Teresa 333
Riedel Franziska 444
Schurian Alexander 222
Schurian Isabella 555
Straka Tobias 111

Du wirst wahrscheinlich gemerkt haben, wie mühsam es ist, zuerst die fünf Personen zu erzeugen und anschließend diese in das Telefonbuch einzuspeichern. Diese Prozedur kannst du auch vom Programm erledigen lassen, indem der Konstruktor der Klasse *Telefonbuch* erweitert wird.

```
public Telefonbuch()
{
    buch = new TreeSet();
    Person[] testdaten = {
        new Person("Straka", "Tobias", "111"),
        new Person("Schurian", "Alexander", "222"),
        new Person("Heider", "Teresa", "333"),
        new Person("Riedel", "Franziska", "444"),
        new Person("Schurian", "Isabella", "555")
    };
    for (int i = 0; i < testdaten.length; i++) {
        neuePerson(testdaten[i]);
    }
}
```

Abbildung 1.19: Der Konstruktor füllt das Telefonbuch mit den Testdaten

Übung 1.20:

Implementiere den in Abbildung 1.19 dargestellten Konstruktor in der Klasse *Telefonbuch*. Überprüfe, ob der Konstruktor nun alle Personen korrekt im Telefonbuch verwaltet.

Füge mit Hilfe der Objektleiste in **BlueJ** noch weitere Personen in das Telefonbuch ein.

1.2.3 Die Klasse Telefonbuch vom Typ *Map* (für Experten)

Nun wird das Telefonbuch als sortierte Sammlung vom Typ *TreeMap* angelegt.

```
import java.util.*;

public class Telefonbuch
{
private TreeMap<String, Person> buch;

    public Telefonbuch()
    {
        buch = new TreeMap<String, Person>();
    }

    public void neuePerson(Person person)
    {
        if (person != null) {
            buch.put(person.gibNachname(), person);
        }
    }

    public void zeigeAllePersonen()
    {
        for (Person person : buch.values()) {
            String nachname = person.gibNachname();
            String vorname = person.gibVorname();
            String telefon = person.gibTelefon();
            System.out.println(nachname + " " + vorname + " " + telefon);
        }
        System.out.println();
    }
}
```

Abbildung 1.20: Die Klasse *Telefonbuch* im Projekt *Telefon1c*

Da nun in dieser Sammlung keine Namen, sondern Personen verwaltet werden, muss die Methode *neuPerson()* entsprechend implementiert werden. Als Wert-

Schlüssel-Paar verwendest du (Nachname/Person), somit kann jedem Nachnamen eindeutig eine Person zugeordnet werden.

Die Methode *zeigeAllePersonen()* durchläuft mit Hilfe eines Iterators die Menge aller Werte und listet die Daten jeder Person auf.

Übung 1.21:

Erstelle in **BlueJ** das Projekt *Telefon1c* und implementiere die Klassen *Person* wie inAbbildung 1.15 und *Telefonbuch* wie in Abbildung 1.20. Untersuche anschließend folgende Probleme:

a) Erzeuge folgende drei Objekte der Klasse *Person* (Straka Tobias 111), (Schurian Alexander 222), (Heider Teresa 333) und ein Objekt der Klasse *Telefonbuch*. Füge mit Hilfe der Methode *neuePerson()* die drei Personen in das Telefonbuch ein und rufe die Methode *zeigeAllePersonen()* auf. In welcher Reihenfolge werden die Daten dieser Personen dargestellt? Begründe dies schriftlich!

b) Erzeuge zwei weitere Personen (Riedel Franziska 444), (Schurian Isabella 555) und füge diese zusätzlich in das Telefonbuch ein. In welcher Reihenfolge werden nun die Daten dieser Personen dargestellt? Was fällt dir über die Anzahl der Personen auf? Begründe schriftlich, warum (Schurian Alexander 222) nicht mehr in der *Map* existiert! (Vergleiche dazu Übung 1.14)

Die Abbildung 1.21 zeigt, dass zwar die Personen in alphabetischer Reihenfolge aufgelistet werden, jedoch fehlt die Person2 (Schurian Alexander 222)

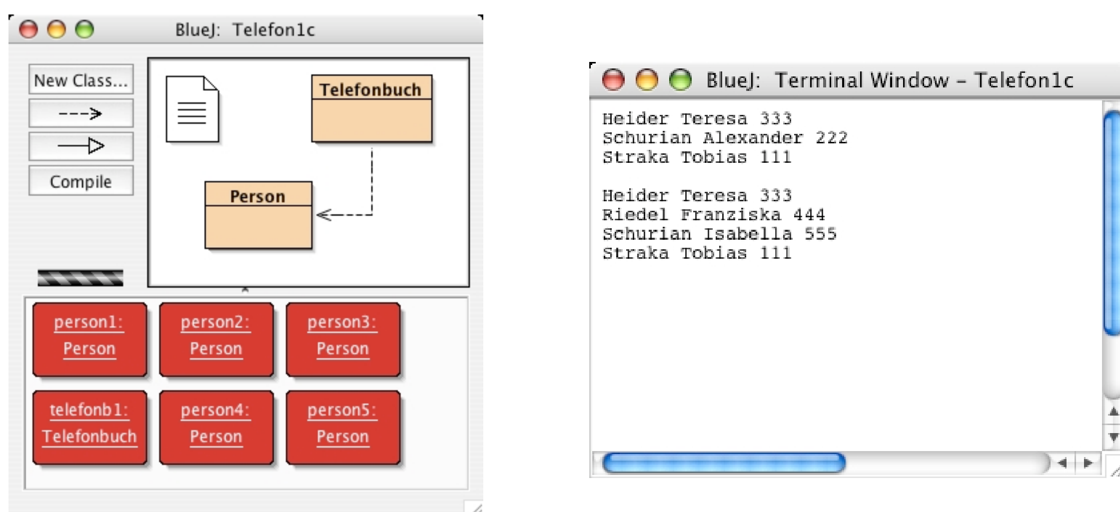

Abbildung 1.21: Die *person2* (Schurian Alexander 222) wird nicht aufgelistet

Um beide Personen (Schurian Alexander 222) und (Schurian Isabella 555) in unserem Telefonbuch zu speichern, wird die Methode *neuePerson()* in der Klasse *Telefonbuch* verändert:

```
public void neuePerson(Person person)
{
    if (person != null) {
        buch.put(person.gibNachname(), person);
        buch.put(person.gibVorname(), person);
    }
}
```

Abbildung 1.22: Erweiterte Methode *neuePerson()* in der Klasse *Telefonbuch*

Übung 1.22:

Erweitere die Methode *neuePerson()* in der Klasse *Telefonbuch*.

Erzeuge folgende fünf Objekte der Klasse *Person* (Straka Tobias 111), (Schurian Alexander 222), (Heider Teresa 333), (Riedel Franziska 444), (Schurian Isabella 555) und ein Objekt der Klasse *Telefonbuch*. Füge mit Hilfe der Methode *neuePerson()* die fünf Personen in das Telefonbuch ein, wobei jedoch nach jedem Einfügen einer neuen Person die jeweils Methode *zeigeAllePersonen()* aufgerufen werden soll.

Vergleiche die Einträge mit der unten gezeigten Tabelle.

Nach Einfügen von:	**zeigt das Terminal:**
Straka Tobias 111	**S**traka Tobias 111 Straka **T**obias 111
Schurian Alexander 222	Schurian **A**lexander 222 **S**churian Alexander 222 **S**traka Tobias 111 Straka **T**obias 111
Heider Teresa 333	Schurian **A**lexander 222 **H**eider Teresa 333 **S**churian Alexander 222 **S**traka Tobias 111 Heider **T**eresa 333 Straka **T**obias 111
Riedel Franziska 444	Schurian **A**lexander 222 Riedel **F**ranziska 444 **H**eider Teresa 333

	Riedel Franziska 444 **S**churian Alexander 222 **S**traka Tobias 111 Heider **T**eresa 333 Straka **T**obias 111
Schurian Isabella 555	Schurian **A**lexander 222 Riedel **F**ranziska 444 **H**eider Teresa 333 Schurian **I**sabella 555 **R**iedel Franziska 444 **S**churian Isabella 555 **S**traka Tobias 111 Heider **T**eresa 333 Straka **T**obias 111

Formuliere schriftlich, warum

a) das Terminal nach jedem weiteren Einfügen einer neuen Person die oben gezeigte Reihenfolge in der *Map* anzeigt,

b) die Person (Schurian Alexander 222) nach dem Einfügen von (Schurian Isabella 555) in der Map nur einmal erscheint.

Der Vorteil, dass nun beide Geschwister Schurian Isabella und Alexander im Telefonbuch vorhanden sind, wurde durch den Nachteil erkauft, dass die meisten Personen nun doppelt vorkommen.

Da jedoch nur ein *Set* eine duplikatenfreie Menge von Elementen darstellt, kannst du die Menge der Werte der *Map* (hier also die Personen) in ein *Set buchSet* umwandeln. Wird nun über *buchSet* iteriert, erhältst du alle Personen ohne Duplikate.

```
public void zeigeAllePersonen()
{
    HashSet<Person> buchSet = new HashSet<Person>(buch.values());
    for (Person person : buchSet) {
        String nachname = person.gibNachname();
        String vorname = person.gibVorname();
        String telefon = person.gibTelefon();
        System.out.println(nachname + " " + vorname + " " + telefon);
    }
    System.out.println();
}
```

Abbildung 1.23: Erweiterung der Methode *zeigeAllePersonen()* in der Klasse *Telefonbuch*

Übung 1.23:

Führe die in Abbildung 1.23 dargestellten Änderungen in der Methode *zeigeAllePersonen()* durch.

Erzeuge anschließend folgende fünf Objekte der Klasse *Person* (Straka Tobias 111), (Schurian Alexander 222), (Heider Teresa 333), (Riedel Franziska 444), (Schurian Isabella 555) und ein Objekt der Klasse *Telefonbuch*. Füge mit Hilfe der Methode *neuePerson()* die fünf Personen in das Telefonbuch ein und rufe die Methode *zeigeAllePersonen()* auf.

Das Terminal sollte folgenden Eintrag (eventuell in anderer Reihenfolge) zeigen:

Heider Teresa 333
Riedel Franziska 444
Schurian Alexander 222
Schurian Isabella 555
Straka Tobias 111

Leider sind die Einträge noch nicht sortiert. Du kennst jedoch bereits die sortierte Sammlung *TreeSet*. Analog Übung 1.13 ersetzt du das Schlüsselwort *HashSet* durch *TreeSet*.

Übung 1.24

Ersetze in der Methode *zeigeAllePersonen()* das Schlüsselwort *HashSet* durch *TreeSet*.

Erzeuge anschließend zwei Objekte der Klasse *Person* und ein Objekt der Klasse *Telefonbuch*. Füge mit Hilfe der Methode *neuePerson()* diese beiden Personen in das Telefonbuch ein und rufe die Methode *zeigeAllePersonen()* auf.

Welches Problem taucht nun auf?

1.2.4 Die Methode *compareTo()*

Bei der *TreeSet*-Collection werden die Elemente sortiert zurückgeliefert. Dazu muss jedes Element der Sammlung die Methode *compareTo()* besitzen, damit zwei beliebige Elemente miteinander verglichen werden können.

Bemerkung:

Seit dem JDK 1.2 wird das *Comparable*-Interface bereits von vielen der eingebauten Klassen implementiert, etwa von *String*, *Character*, *Double*, usw.

Die natürliche Ordnung einer Menge von Elementen ergibt sich nun, indem man alle Elemente paarweise miteinander vergleicht und dabei jeweils das kleinere vor das größere Element stellt.
(aus „Handbuch der Java-Programmierung“ von G. Krüger)

Da du nur Strings in den Sammlungen verwaltet hast, brauchtest du dir also um die natürliche Ordnung keine Gedanken machen, da in der Klasse *String* das *Comparable*-Interface implementiert ist. Werden jedoch in diesen Sammlungen beliebige Objekte z.B. Objekte einer Klasse *Person* verwaltet, musst du mit der Methode *compareTo*() eine Ordnung selbst erzeugen.

compareTo() liefert einen Wert kleiner 0, wenn das aktuelle Element vor dem zu vergleichenden liegt,

liefert einen Wert größer 0, wenn das aktuelle Element hinter dem zu vergleichenden liegt,

liefert den Wert 0, wenn das aktuelle Element und das zu vergleichenden gleich sind..

An Hand des zurückgegebenen Wertes kann nun die Ordnung erstellt werden.

```
ublic class Person implements Comparable
{

……… weitere Anweisungen ………

    public int compareTo(Object jene)
    {
        Person jenePerson = (Person) jene;
        int vergleich = nachname.compareTo(jenePerson.gibNachname());
        if (vergleich != 0) {
            return vergleich;
        }
        vergleich = vorname.compareTo(jenePerson.gibVorname());
        if (vergleich != 0) {
            return vergleich;
        }
        vergleich = telefon.compareTo(jenePerson.gibTelefon());
        return vergleich;
    }
}
```

Abbildung 1.24: Die Methode *compareTo()* in der Klasse *Person*

Übung 1.25:

Implementiere die Methode *compareTo()* in der Klasse *Person*.

Erzeuge folgende fünf Objekte der Klasse *Person* (Straka Tobias 111), (Schurian Alexander 222), (Heider Teresa 333), (Riedel Franziska 444), (Schurian Isabella 555) und ein Objekt der Klasse *Telefonbuch*. Füge mit Hilfe der Methode *neuePerson()* die fünf Personen in das Telefonbuch ein und rufe die Methode *zeigeAllePersonen()* auf.

Das Terminal sollte folgenden Eintrag zeigen:

Heider Teresa 333
Riedel Franziska 444
Schurian Alexander 222
Schurian Isabella 555
Straka Tobias 111

Du wirst wahrscheinlich gemerkt haben, wie mühsam es ist, zuerst die fünf Personen zu erzeugen und anschließend diese in das Telefonbuch einzuspeichern. Du kannst diese Prozedur auch vom Programm erledigen lassen, indem der Konstruktor der Klasse *Telefonbuch* erweitert wird.

```
public Telefonbuch()
{
    buch = new TreeMap<String, Person>();
    Person[] testdaten = {
        new Person("Straka", "Tobias", "111"),
        new Person("Schurian", "Alexander", "222"),
        new Person("Heider", "Teresa", "333"),
        new Person("Riedel", "Franziska", "444"),
        new Person("Schurian", "Isabella", "555")
    };
    for (int i = 0; i < testdaten.length; i++) {
        neuePerson(testdaten[i]);
    }
}
```

Abbildung 1.25: Der Konstruktor füllt das Telefonbuch mit den Testdaten

Übung 1.26:

Implementiere den in Abbildung 1.25 dargestellten Konstruktor in der Klasse *Telefonbuch*. Überprüfe, ob der Konstruktor nun alle Personen korrekt im Telefonbuch verwaltet.

Füge mit Hilfe der Objektleiste in **BlueJ** noch weitere Personen in das Telefonbuch ein.