Programmierkurs Java

Dr.-Ing. Dietrich Boles

Aufgaben zu UE 13 – Vererbung

(Stand 22.03.2019)

Aufgabe 1:

Stellen Sie sich vor, Sie benötigen eine Klasse, die eine Menge von Zahlen repräsentiert, d.h. Objekte dieser Klasse sollen eine bestimmte Anzahl von int-Werten speichern können, jeden int-Wert maximal einmal. Sie durchsuchen die Klassenbibliothek und finden eine Klasse `Container`, die genau das von Ihnen gewünschte Protokoll definiert. Ein Unterschied zu Ihrer `Menge`-Klasse - und das ist der einzige Unterschied - besteht darin, dass Container-Objekte int-Werte auch mehrfach speichern können. Die Klasse Container habe folgende Gestalt:

```
class Container {
     int next;
     int[] elems;

     // initialisiert ein Container-Objekt, das max. size Werte
     // speichern kann
     Container(int size) {
           this.elems = new int[size];
           this.next = 0;
     }

     // voll?
     boolean isFull() {
           return this.next == this.elems.length;
     }

     // speichert value ab, auch wenn der Wert bereits gespeichert ist;
     // liefert true, wenn der Wert abgespeichert wurde
     boolean addElement(int value) {
           if (isFull()) {
                 return false;
           }
           this.elems[this.next++] = value;
           return true;
     }

     // entfernt alle Vorkommen von value aus dem Speicher
     // liefert false, wenn der Wert nicht vorhanden war
     boolean removeElement(int value) {
           boolean result = false;
           int i = 0;
           while (i < this.next) {
                 if (this.elems[i] == value) {
                       result = true;
                       for (int j = i; j < this.next - 1; j++) {
```

```
                        this.elems[j] = this.elems[j + 1];
                    }
                    this.next--;
                } else {
                    i++;
                }
            }
            return result;
        }

        // ueberprueft, ob value bereits gespeichert ist
        boolean existsElement(int value) {
            for (int i = 0; i < this.next; i++) {
                if (this.elems[i] == value) {
                    return true;
                }
            }
            return false;
        }
    }
```

Aufgabe: Implementieren Sie Ihre Klasse *Menge*, indem Sie sie von der Klasse *Container* ableiten und die geerbten Attribute und Methoden nutzen!

Hinweise zu den Aufgaben 2, 3, 4, 14:

Vorgegeben für die Aufgaben 2, 3, 4 und 14 ist folgende Klasse `Stack`:

```
class Stack {

    int[] store; // zum Speichern von Daten
    int current; // aktueller Index

    Stack(int size) {
        this.store = new int[size];
        this.current = -1;
    }

    boolean isFull() {
        return this.current == this.store.length - 1;
    }

    boolean isEmpty() {
        return this.current == -1;
    }

    boolean push(int value) {
        if (!this.isFull()) {
            this.store[++this.current] = value;
            return true;
        }
        return false;
    }

    int pop() {
        if (!this.isEmpty()) {
            return this.store[this.current--];
        }
        return 0; // Fehler! Wird spaeter via Exceptions gehandhabt
    }
}
```

Stacks sind Behälter-Klassen, auf denen mittels einer Methode `push` Werte ausschließlich oben drauf abgelegt werden können und mit Hilfe einer Methode `pop` der jeweils oberste Wert ermittelt und vom Stack entfernt werden kann.

Aufgabe 2:

Ihnen steht die oben definierte Klasse `Stack` zur Verfügung. Sie benötigen jedoch für eine bestimmte Anwendung einen Stack, bei dem Sie zu bestimmten Zeitpunkten die Reihenfolge der auf dem Stack gespeicherten Elemente umdrehen können, d.h. das oberste Element kommt nach ganz unten und dieses nach oben und die dazwischen liegenden Elemente werden gleichfalls getauscht.

Entwickeln Sie eine Klasse `ResortStack` mit einer Methode `void resort()` , die eine entsprechende Umsortierung vornimmt. Nutzen Sie dabei die Möglichkeit, die Klasse `ResortStack` von der obigen Klasse `Stack` abzuleiten und deren Attribute und Methoden zu erben. Die Klasse `Stack` selbst darf nicht verändert werden!

Schreiben Sie weiterhin ein kleines Programm zum Testen der Klasse `ResortStack`.

Aufgabe 3:

Ihnen steht die oben definierte Klasse `Stack` zur Verfügung. Sie benötigen jedoch für eine bestimmte Anwendung einen Stack, bei dem Sie zusätzlich abfragen können wollen, wie viele erfolgreiche push-Operationen für diesen Stack erfolgt sind.

Nutzen Sie die Möglichkeit, eine entsprechende Klasse `CountStack` von der obigen Klasse `Stack` abzuleiten und deren Attribute und Methoden zu erben. Die Klasse `Stack` selbst darf nicht verändert werden! Definieren Sie in der Klasse `CountStack` eine zusätzliche Methode `int getNumberOfValidPushOperations()`, die eine entsprechende Abfrage ermöglicht. Die geerbte Methode `push` muss dabei adäquat überschrieben werden.

Testen Sie Ihr Programm mit folgendem Testprogramm:

```
public class UE13Aufgabe3 {

	// Testprogramm
	public static void main(String[] args) {
		CountStack stack = new CountStack(4);
		while (!stack.isFull()) {
			stack.push(IO.readInt("Eingabe: "));
		}
		stack.push(47);
		while (!stack.isEmpty()) {
			System.out.println(stack.pop());
		}
		while (!stack.isFull()) {
			stack.push(IO.readInt("Eingabe: "));
		}
		System.out.println(stack.getNumberOfValidPushOperations());
		// Ausgabe sollte 8 sein!
	}
}
```

Aufgabe 4:

Ihnen steht die oben definierte Klasse `Stack` zur Verfügung. Sie benötigen jedoch für eine bestimmte Anwendung einen Stack, bei dem zu bestimmten Zeitpunkten der Stack sortiert werden soll

Entwickeln Sie eine Klasse `SortStack` mit einer Methode `void sort()` , die eine entsprechende Sortierung vornimmt. Nutzen Sie dabei die Möglichkeit, die Klasse `SortStack` von der obigen Klasse `Stack` abzuleiten und deren Attribute und Methoden zu erben. Die Klasse `Stack` selbst darf nicht verändert werden!

Schreiben Sie weiterhin ein kleines Programm zum Testen der Klasse `SortStack`.

Aufgabe 5:

Gegeben sei folgender Source-Code:

```
class Grossvater {
	int x = 3;

	int y = -4;
}

class Vater extends Grossvater {
	float x = 4.5F;

	int z;

	public Vater(int z) {
		this.z = z;
	}
}

class Sohn extends Vater {
	long a;

	double x = -18.5;

	public Sohn(long a) {
		super(5);
		this.a = a;
	}
}
```

Leiten Sie von der Klasse Sohn eine Klasse `Enkel` ab, die eine Methode besitzt, in der die Werte aller Attribute, die ein Objekte der Klasse `Enkel` besitzt (auch die geerbten!), addiert werden und die die Summe auf den Bildschirm ausgibt. Schreiben Sie ein kleines Testprogramm!

Hinweise zu den Aufgabe 6 bis 9:

Vorgegeben für die Aufgaben 6 bis 9 sind folgende Klassen `Element` und `List`. Sie implementieren die Datenstruktur einer „verketteten Liste“. Eine verkettete Liste ist eine dynamische Datenstruktur, die eine Speicherung von miteinander in Beziehung

stehenden Objekten erlaubt. Die Anzahl der Objekte ist im Vorhinein nicht bestimmt. Die Liste wird durch Referenzen auf die jeweils folgende(n) Elemente realisiert.

```
class Element {
	int value; // Speicher fuer einen Wert
	Element next; // Referenz auf das folgende Element (oder null fuer Ende)

	Element(int v, Element n) {
		this.value = v;
		this.next = n;
	}
}

class List {
	Element first; // erstes Element der Liste
	Element last; // letztes Element der Liste

	List() {
		this.first = null;
		this.last = null;
	}

	void append(int i) {
		Element elem = new Element(i, null);
		if (this.first == null) { // erstes Element in der Liste
			this.first = elem;
			this.last = elem;
		} else { // Anhaengen des Elementes am Ende
			this.last.next = elem;
			this.last = elem;
		}
	}

	void print() {
		Element elem = this.first;
		while (elem != null) {
			System.out.print(elem.value);
			if (elem.next != null) {
				System.out.print(" -> ");
			}
			elem = elem.next;
		}
		System.out.println();
	}
}
```

Aufgabe 6:

Leiten Sie von der oben definierten Klasse *List* eine Klasse *ExtList* ab.

- Überschreiben Sie die Methode `append` derart, dass kein int-Wert mehrfach in die Liste eingetragen werden kann.
- Definieren Sie eine zusätzliche Methode `void prepend(int v)`, mit der ein int-Wert vorne in die Liste eingetragen werden kann.
- Definieren Sie weiterhin eine zusätzliche Methode `void change(int v)`, welche folgendes bewirkt:

- Befindet sich der übergebene Parameterwert bereits in der Liste, so wird er entfernt.
- Befindet sich der übergebene Parameterwert noch nicht in der Liste, so wird er am Anfang der Liste eingefügt.

Beispiel:

Die Liste `5 -> 4 -> 2 -> 7 -> 1` wird durch den Aufruf der Methode `change(4)` zu der Liste `5 -> 2 -> 7 -> 1` und anschließend durch Aufruf von `change(8)` zu der Liste `8 -> 5 -> 2 -> 7 -> 1`

Aufgabe 7:

Leiten Sie von der oben definierten Klasse `List` eine Klasse `RList` ab, die eine zusätzliche Methode `remove` definiert, welche aus einer Liste jedes Element entfernt, das den gleichen Wert hat wie sein Vorgängerelement.

Beispiel: Die Liste

```
5 -> 4 -> 4 -> 5 -> 2 -> 2 -> 2 -> 4
```

wird durch den Aufruf der Methode `remove` zu der Liste

```
5 -> 4 -> 5 -> 2 -> 4
```

Aufgabe 8:

Leiten Sie von der oben definierten Klasse `List` eine Klasse `AList` ab, die eine zusätzliche Methode `void ansEndeSchieben(int v)` definiert. Ein Aufruf der Methode soll bewirken, dass alle Elemente der Liste mit dem Wert des aktuellen Parameters `v` an das Ende der Liste platziert werden.

Beispiel: Die Liste

```
5 -> 4 -> 3 -> 4 -> 4 -> 2 -> 2 -> 1
```

wird durch den Aufruf der Methode `ansEndeSchieben(4)` zu der Liste

```
5 -> 3 -> 2 -> 2 -> 1 -> 4 -> 4 -> 4
```

Aufgabe 9:

Leiten Sie von der oben definierten Klasse `List` eine Klasse `VList` ab, die eine zusätzliche Methode `void verdoppeln(int v)` definiert. Ein Aufruf der Methode soll bewirken, dass alle Elemente der Liste mit dem Wert des aktuellen Parameters `v` verdoppelt werden.

Beispiel: Die Liste

```
5 -> 4 -> 3 -> 4 -> 4 -> 2 -> 2 -> 1
```

wird durch den Aufruf der Methode `verdoppeln(4)` zu der Liste

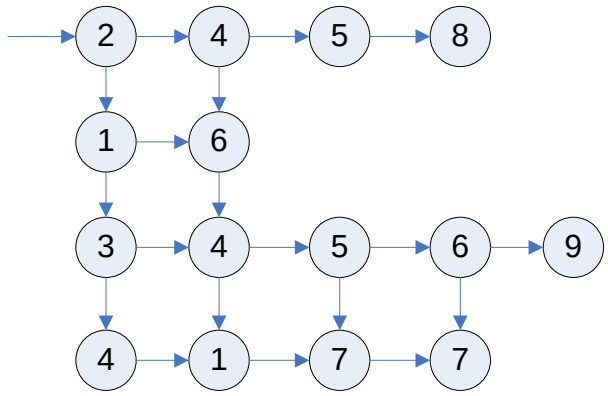
```
5 -> 4 -> 4 -> 3 -> 4 -> 4 -> 4 -> 4 -> 2 -> 2 -> 1
```

Aufgabe 10:

In den Aufgaben 6 bis 9 haben Sie verkettete Listen kennengelernt. Eine verkettete Liste ist eine eindimensionale dynamische Datenstruktur, die eine Speicherung von einer im Vorhinein nicht bestimmten Anzahl von miteinander in Beziehung stehenden Werten erlaubt. Sie wird durch eine Menge an Elementen realisiert, die neben den eigentlichen Werten Referenzen auf das jeweils folgende Element speichern.

In dieser Aufgabe geht es um die Erweiterung einer verketteten Liste zu einer verketteten Matrix. Eine „verkettete Matrix“ ist eine zweidimensionale dynamische Datenstruktur, die sich von einer verketteten Liste dadurch unterscheidet, dass jedes Element zusätzlich eine Referenz auf das darunter liegende Element speichert. Die folgende Abbildung zeigt beispielhaft die Umsetzung einer int-Matrix (links) als verkettete Matrix (rechts).

```
{{2, 4, 5, 8},
 {1, 6},
 {3, 4, 5, 6, 9},
 {4, 1, 7, 7}
}
```



Die folgende Klasse `Elem` repräsentiert dabei die einzelnen Elemente. Ein `Elem`-Objekt speichert jeweils einen int-Wert (Attribut `wert`) und referenziert das rechtsstehende Element (Attribut `rechts`) und das darunter liegende Element (Attribut `unten`). Wenn sich rechts von bzw. unter einem Element kein anderes Element befindet, ist der Wert der Attribute `rechts` bzw. `unten` gleich null.

```
class Elem {
      int wert;
      Elem rechts;
      Elem unten;

      Elem(int w, Elem r, Elem u) {
            this.wert = w;
            this.rechts = r;
            this.unten = u;
      }

      Elem (int w) {
            this(w, null, null);
      }
}
```

Ihre Aufgabe besteht nun darin, aufbauend auf den gemachten Erläuterungen einen Ausschnitt einer entsprechenden Klasse `VerketteteMatrix` zu implementieren.

```
// vorgegeben
class Elem {
	int wert;
	Elem rechts;
	Elem unten;

	Elem(int w, Elem r, Elem u) {
		this.wert = w;
		this.rechts = r;
		this.unten = u;
	}

	Elem(int w) {
		this(w, null, null);
	}
}

public class VerketteteMatrix {

	private Elem linksOben; // Referenz auf das linke obere Element

	/**
	 * Erzeugt ein VerketteteMatrix-Objekt, das der uebergebenen Matrix
	 * entspricht
	 *
	 * @param matrix
	 *            die Matrix, die in ein VerketteteMatrix-Objekt
	 *            überführt werden soll; Voraussetzungen:
	 *            matrix != null && matrix.length > 0 &&
	 *            matrix[r].length > 0 fuer alle Reihen r
	 */
	public VerketteteMatrix(int[][] matrix) { }

	/**
	 * Liefert den Wert des Elementes in der angegebenen Reihe und Spalte
	 *
	 * @param reihe
	 *            Reihe des Elementes
	 * @param spalte
	 *            Spalte des Elementes
	 * @return Wert des Elementes in der angegebenen Reihe und Spalte
	 */
	public int getWert(int reihe, int spalte) {    }

	/**
	 * Aendert den Wert des Elementes in der angegebenen Reihe und
	 * Spalte. Bei "Luecken" in der Matrix werden diese mit Nullen
	 * aufgefuellt!
	 *
	 * @param wert
	 *            neuer Wert des Elementes
	 * @param reihe
	 *            Reihe des Elementes
	 * @param spalte
	 *            Spalte des Elementes
	 */
	public void setWert(int wert, int reihe, int spalte) {
	   // Implementierung nicht von Ihnen gefordert
	}

	// vorgegeben
	public void print() {
		Elem elemReihe = linksOben;
```

```
            while (elemReihe != null) {
                  Elem elemSpalte = elemReihe;
                  while (elemSpalte != null) {
                        System.out.print(elemSpalte.wert + " ");
                        elemSpalte = elemSpalte.rechts;
                  }
                  System.out.println();
                  elemReihe = elemReihe.unten;
            }
      }

      // vorgegebenes Testprogramm
      public static void main(String[] args) {
            int[][] m = {     { 2, 4, 5, 8 },
                              { 1, 6 },
                              { 3, 4, 5, 6, 9 },
                              { 4, 1, 7, 7 }
                        };
            VerketteteMatrix matrix = new VerketteteMatrix(m);
            matrix.print();
      }

}
```

Genauer formuliert, sollen Sie den Konstruktor sowie die Methode `getWert` der Klasse `VerketteteMatrix` gemäß den obigen Ausführungen implementieren.

Schauen Sie sich bitte auch die vorgegebene Implementierung der Methode `print` an, die bewirken soll, dass das vorgegebene Testprogramm die folgende Ausgabe produziert:

```
2 4 5 8
1 6
3 4 5 6 9
4 1 7 7
```

Aufgabe 11:

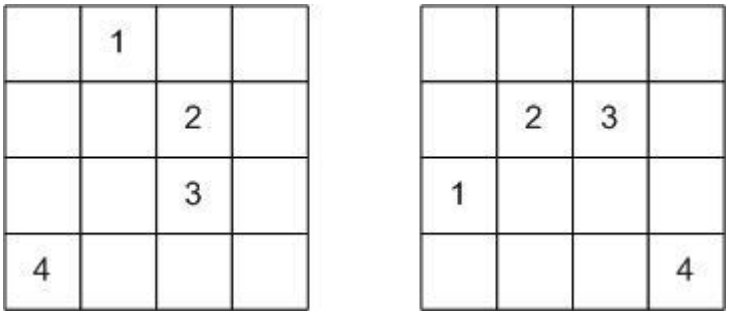
Stellen Sie sich vor, jemand hat ein bestimmtes Spiel implementiert. Dieses besteht u. a. aus folgenden Klassen:

```
class Spielfigur { ...
}

class Spielfeld {

      int size;

      Spielfigur[][] figuren;

      Spielfeld(int size) {
            this.size = size;
            this.figuren = new Spielfigur[this.size][this.size];
            for (int r = 0; r < this.size; r++) {
                  for (int s = 0; s < this.size; s++) {
                        figuren[r][s] = null;
                  }
            }
      }

      int getSize() {
            return size;
```

```
    }

    void setSpielfigur(Spielfigur figur, int reihe, int spalte) {
        if (reihe >= 0 && reihe < this.size && spalte >= 0
                && spalte < this.size && figur != null) {
            this.figuren[reihe][spalte] = figur;
        }
    }

    Spielfigur getSpielfigur(int reihe, int spalte) {
        if (reihe < 0 || reihe >= this.size ||
            spalte < 0 || spalte >= this.size ||
            this.figuren[reihe][spalte] == null) {
            return null;
        }
        return this.figuren[reihe][spalte];
    }

    // weitere fuer die Aufgabe irrelevante Methoden
}
```

Sie möchten nun ein ziemlich ähnliches Spiel implementieren und wollen dazu natürlich die zur Verfügung stehenden Kassen nutzen. Die Klasse `Spielfigur` können Sie ohne Einschränkungen benutzen, die Klasse `Spielfeld` im Prinzip auch, allerdings brauchen Sie eine zusätzliche Methode, mit der das Spielfeld um 90 Grad nach links (also gegen den Uhrzeigersinn) gedreht werden kann.

Aufgabe: Leiten Sie von der Klasse `Spielfeld` eine Klasse `MeinSpielfeld` ab, die eine zusätzliche Methode zum Drehen des Spielfeldes um 90 Grad nach links zur Verfügung stellt. Sie dürfen die Klasse `Spielfeld` nicht verändern.



Spielfeld um 90 Grad nach links gedrehtes Spielfeld

Aufgabe 12:

Schnappen Sie sich ein Biologiebuch und entwerfen Sie eine Klassenhierarchie für die Tierwelt. Die Klassenhierarchie sollte mindestens 20 Klassen bzw. Interfaces enthalten. Berücksichtigen Sie u.a. folgende Phänomene:

- Tiere können unterschiedlicher Art sein (Säugetiere, Vögel, Reptilien, ...)
- Tiere können unterschiedliche Lebensräume haben (Land, Wasser, Luft, ... (auch Kombinationen!))

- Tiere können unterschiedliche Fortbewegungsarten haben (Laufen, Schwimmen, Fliegen, ... (auch Kombinationen!))
- Tiere können unterschiedliche Fortpflanzungsmethoden besitzen
- Tiere können unterschiedliche Ernährungsmethoden besitzen (Fleischfresser, Pflanzenfresser, Allesfresser)

Begründen Sie Ihre Entscheidungen! Begründen Sie jeweils, weshalb Sie sich für Klassen, abstrakte Klassen, Interfaces bzw. Aggregation entschieden haben! Ordnen Sie den Klassen/Interfaces geeignete Methoden mit geeigneten Parametern zu!

Hinweise:

- Es gibt keine eindeutige Lösung!
- Sie brauchen die Methoden nicht zu implementieren!
- Stellen Sie die Klassenhierarchie auch graphisch dar (besserer Überblick!)

Aufgabe 13:

Eine Klasse `RundFunkEmpfangsGeraet` sei folgendermaßen definiert:

```
class RundfunkEmpfangsGeraet {
     int lautstaerke;
     boolean eingeschaltet;

     RundfunkEmpfangsGeraet() {
     }

     /**
      * Verändere Lautstärke um x (nach oben oder unten, je nach
      * Vorzeichen von x).
      */
     void volume(int x) {
     }

     /** Erhöhe Lautstärke um 1. */
     void lauter() {
     }

     /** Liefere Lautstärke zurück. */
     int getLautstaerke() {
     }

     /** Vermindere Lautstärke um 1. */
     void leiser() {
     }

     /** Schalte ein. */
     void an() {
     }

     /** Schalte aus. */
     void aus() {
     }

     /** An oder aus? */
     boolean istAn() {
     }
}
```

Teilaufgabe (1): Implementieren Sie die Methoden dieser Klasse adäquat. Ein neues Objekt der Klasse `RundfunkEmpfangsGeraet` soll Lautstärke 0 haben und ausgeschaltet sein. Einstellungen an einem Empfangsgerät sollen nur verändert werden können, wenn das Gerät an ist.

Teilaufgabe (2): Leiten Sie von der Klasse `RundfunkEmpfangsGeraet` zwei Klassen `Radio` und `Fernseher` ab. Bei Radios und Fernsehern soll es sich dabei um Empfangsgeräte handeln, die zusätzlich jeweils noch folgende Attribute besitzen bzw. Methoden bereitstellen.

Für die Klasse `Radio` soll gelten:

- Diese Klasse besitzt ein Attribut `double frequenz`, das über die Methode `void waehleSender(double newFrequenz)` verändert werden kann, falls das Radio eingeschaltet ist.
- Ein neues `Radio`-Objekt ist auf die Frequenz 87.5 MHz eingestellt.
- Über die Methode `String makeString()` sollen die Einstellungen des Radios zurückgeben werden, in der Form `Radio: Sender = 104.1 Lautstaerke = 11, Radio ist an.`

Für die Klasse `Fernseher` soll gelten:

- Die Klasse `Fernseher` besitzt ein Attribut `int kanal`, das über die Methode `void waehleKanal(int newKanal)` verändert werden kann, falls der Fernseher eingeschaltet ist.
- Ein neues `Fernseher`-Objekt ist auf den Kanal 1 eingestellt.
- Über die Methode `String makeString()` sollen die Einstellungen des Fernsehers zurückgeben, in der Form `Fernseher: Kanal = 3 Lautstaerke = 20, Fernseher ist an.`

Teilaufgabe (3): Schreiben Sie ein kleines Testprogramm.

Aufgabe 14:

Ihnen steht die oben (vor Aufgabe 2) definierte Klasse `Stack` zur Verfügung. Sie benötigen jedoch einen Stack, bei dem sichergestellt ist, dass ein `int`-Wert nicht mehrfach im Stack vorkommt. Leiten Sie daher eine Klasse `UniqueStack` von der Klasse `Stack` ab, in der die Methode `push` derart überschrieben wird, dass die Anforderung erfüllt ist.

Aufgabe 15:

In einem Kreis stehen *anzahl* Kinder (durchnumeriert von 1 bis *anzahl*). Mit Hilfe eines *silbe*-silbigen Abzählreims wird das jeweils *silbe*-te unter den noch im Kreis befindlichen Kindern ausgeschieden, bis kein Kind mehr im Kreis steht. Ihre Aufgabe besteht darin, ein Java-Programm, das nach Vorgabe von *anzahl* (positive Zahl) und *silbe* (positive Zahl) die Nummern der Kinder in der Reihenfolge ihres Ausscheidens angibt.

Beispiel*:* Für *anzahl*=6 und *silbe*=5 ergibt sich die Folge 5, 4, 6, 2, 3, 1.

Hinweis: Im Rahmen der Vorlesungen/Übungen wurde dieses Programm bereits mit Hilfe von Arrays sowie ArrayList-Objekten gelöst. Bei der jetzt gesuchten Lösung sollen verkettete Listen eingesetzt werden.

Der folgende Programmrahmen sei dabei vorgegeben:

```
class Kind {
     private int nummer;       // Nummer des Kindes
     private Kind nachfolger;  // Nachfolgekind im Kreis

     Kind(int nummer, Kind naechfolger) {
           this.nummer = nummer;
           this.nachfolger = naechfolger;
     }

     int getNummer() { return this.nummer;    }

     Kind getNachfolger() { return this.nachfolger; }

     void setNachfolger(Kind nachfolger) {
           this.nachfolger = nachfolger;
     }
}

class Kreis {
     int anzahl;     // aktuelle Anzahl an Kindern im Kreis
     Kind aktuelles; // Kind vor dem zuletzt ausgeschiedenen Kind

     // Vorbedingung: anzahl >= 1
     Kreis(int anzahl) {
           this.anzahl = anzahl;
           Kind kind = new Kind(anzahl, null);
           this.aktuelles = kind;
           for (int i = anzahl - 1; i >= 1; i--) {
                 kind = new Kind(i, kind);
           }
           this.aktuelles.setNachfolger(kind);
     }

     boolean istLeer() {     return this.anzahl == 0;      }

     // Vorbedingung: silbe >= 1
     Kind abzaehlen(int silbe) {
           // Implementierung: Ihre Aufgabe
     }

     void print() {
           // Implementierung: Ihre Aufgabe
     }
}

public class Reim {
     public static void main(String[] args) {
           final int anzahl = 6; // Konstante beliebig >= 1
           final int silbe = 5;  // Konstante beliebig >= 1
           Kreis kreis = new Kreis(anzahl);
           kreis.print();
           while (!kreis.istLeer()) {
                 Kind ausgeschieden = kreis.abzaehlen(silbe);
                 System.out.println("raus: " + ausgeschieden.getNummer());
                 kreis.print();
```

```
            }
        }
}
```

Ihre konkrete Aufgabe besteht in der Implementierung der fehlenden Methoden *abzaehlen* und *print* der Klasse *Kreis*. Dabei müssen Sie mit den vorgegebenen Datenstrukturen und Methoden arbeiten, dürfen sie nicht ändern und dürfen keine weiteren Attribute oder Klassen hinzufügen.

a) **`Kind abzaehlen(int silbe)`**: Liefert ausgehend von dem aktuellen Kind das *silbe*-te sich noch im Kreis befindliche Kind.
b) **`void print()`**: Gibt die Nummern der sich noch im Kreis befindenden Kinder startend mit dem aktuellen Kind hintereinander auf die Konsole aus. Die Nummern werden dabei durch die Zeichenkette `->` getrennt.

Das angegebene Programm sollte folgende Ausgabe erzeugen:

```
6->1->2->3->4->5
raus: 5
4->6->1->2->3
raus: 4
3->6->1->2
raus: 6
3->1->2
raus: 2
1->3
raus: 3
1
raus: 1
Kreis ist leer
```

Aufgabe 16:

Gegeben sind folgende Klassen `Element` und `List`. Sie implementieren die Datenstruktur einer „doppelt verketteten Liste“. Eine doppelt verkettete Liste ist eine dynamische Datenstruktur, die eine Speicherung von miteinander in Beziehung stehenden Objekten erlaubt. Die Anzahl der Objekte ist im Vorhinein nicht bestimmt. Die Liste wird durch Referenzen eines Elementes auf das jeweils folgende und das jeweils vorhergehende Element realisiert.

```
class Element {
  int value;    // Speicher fuer einen Wert
  Element next; // Referenz auf das folgende Element (oder null fuer Ende)
  Element prev; // Referenz auf das vorhergehende Element (oder null fuer
                // Anfang)

  Element(int v, Element n, Element p) {
    this.value = v;
    this.next = n;
    this.prev = p;
  }
}

class List {
  Element first; // erstes Element der Liste
  Element last;  // letztes Element der Liste
```

```
  List() {
    this.first = null;
    this.last = null;
  }

  void append(int i) {
    Element elem = new Element(i, null, this.last);
    if (this.first == null) { // erstes Element in der Liste
      this.first = elem;
      this.last = elem;
    } else { // Anhaengen des Elementes am Ende
      this.last.next = elem;
      this.last = elem;
    }
  }

  void print() {
    Element elem = this.first;
    while (elem != null) {
      System.out.print(elem.value);
      if (elem.next != null) {
        System.out.print(" -> ");
      }
      elem = elem.next;
    }
    System.out.println();
  }
}
```

Aufgabe: Leiten Sie von der oben definierten Klasse `List` eine Klasse `RDList` ab, die eine zusätzliche Methode `void reverseDuplicate()` definiert, welche die Liste in umgekehrter Reihenfolge verdoppelt, d.h. jeder Wert, der sich bisher in der Liste befindet, wird in der Reihenfolge von hinten nach vorne erneut hinten angehängt.

Beispiel: Die Liste

```
5 -> 4 -> 3 -> 4 -> 2 -> 1
```

wird durch den Aufruf der Methode `reverseDuplicate` zu der Liste

```
5 -> 4 -> 3 -> 4 -> 2 -> 1 -> 1 -> 2 -> 4 -> 3 -> 4 -> 5
```

Aufgabe 17:

Gegeben sind folgende Klassen `Node` und `Tree`. Sie implementieren analog zur durch die Vorlesung bekannten Datenstruktur der verketteten Liste einen verketteten binären Suchbaum. Ein binärer Baum (Tree) besteht aus einem einzelnen (Wurzel-)Knoten (Node). Ein Knoten speichert einen Wert und enthält wiederum Referenzen auf einen linken und einen rechten binären Baum. Man spricht von einem binären Suchbaum, wenn die Werte des linken Teilbaums eines Knotens nur kleiner (oder gleich) und die des rechten Teilbaums nur größer (oder gleich) als der Wert des Knotens selbst sind.

```
class Node {
  int value;
  Tree left, right;
```

```
  public Node(int value, Tree tree) {
    this.value = value;
    this.left = tree.createTree();
    this.right = tree.createTree();
  }
}

class Tree {
  protected Node node;

  public Tree() {
    node = null;
  }

  protected Tree createTree() {
    return new Tree();
  }

  public boolean isEmpty() {
    return node == null;
  }

  public void insert(int x) {
    if (isEmpty())
      node = new Node(x, this);
    else if (x < node.value)
      node.left.insert(x);
    else
      node.right.insert(x);
  }

  public String toString() {
    if (isEmpty())
      return "";
    else {
      String s = ":" + node.value + ";";
      return node.left + s + node.right;
    }
  }
}
```

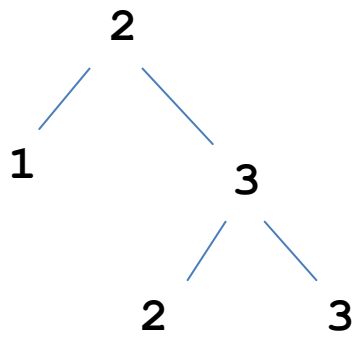
Aufgabe: Leiten Sie von der oben definierten Klasse `Tree` eine Klasse `SearchTree` ab, die eine zusätzliche Methode `public boolean contains(int x)` implementiert, die genau dann *true* liefert, wenn der als Parameter übergebene Wert in irgendeinem Knoten des Baumes gespeichert ist. Überlegen Sie, ob weitere Methoden notwendig sind und implementieren Sie diese gegebenenfalls auch. Die Klassen `Tree` und `Node` dürfen nicht verändert werden!

Beispiel: Das folgende Programm

```
public class Baeume {
  public static void main(String[] args) {
    SearchTree t = new SearchTree();
    t.insert(2);
    t.insert(3);
    t.insert(1);
    t.insert(3);
    t.insert(2);
    System.out.println(t);
    System.out.println(t.contains(1));
    System.out.println(t.contains(4));
```

```
  }
}
```

baut folgenden binären Suchbaum auf



und erzeugt folgende Ausgabe

```
:1;:2;:2;:3;:3;
true
false
```

Aufgabe 18:

Leiten Sie von der schon in Aufgabe 6-9 definierten Klasse `List` eine Klasse `DList` ab, die eine zusätzliche Methode `void duplicate()` definiert, welche die Liste verdoppelt, d.h. jeder Wert, der sich bisher in der Liste befindet, wird in derselben Reihenfolge erneut hinten angehängt.

Beispiel: Die Liste

```
5 -> 4 -> 4 -> 5 -> 2 -> 4
```

wird durch den Aufruf der Methode `duplicate` zu der Liste

```
5 -> 4 -> 4 -> 5 -> 2 -> 4 -> 5 -> 4 -> 4 -> 5 -> 2 -> 4
```

Aufgabe 19:

Gegeben sei folgende aus der Vorlesung/Übung bekannte Klasse **`Stack`**:

```
public class Stack {

  private int[] buffer; // zum Speichern von Daten
  private int current; // aktueller Index

  public Stack(int size) {
    this.buffer = new int[size];
    this.current = -1;
  }

  public int getSize() {
```

```
    return this.buffer.length;
  }

  public boolean isFull() {
    return this.current == this.buffer.length - 1;
  }

  public boolean isEmpty() {
    return this.current == -1;
  }

  public void push(int value) {
    if (!this.isFull()) {
      this.buffer[++this.current] = value;
    }
  }

  public int pop() {
    if (this.isEmpty()) {
      return -1;
    }
    return this.buffer[this.current--];
  }
}
```

Leiten Sie von der Klasse **`Stack`** eine neue Klasse **`ToTopStack`** ab. Die Klasse **`ToTopStack`** soll eine weitere Methode „**`public boolean toTop(int value)`**“ mit folgender Semantik implementieren:

Ist der übergebene Parameterwert aktuell nicht im aufgerufenen Stack vorhanden, soll nach Aufruf der Methode der Zustand des Stacks unverändert sein und die Methode liefert den Wert **`false`**. Ist der Parameterwert ein (oder mehrfach) im Stack vorhanden, wird der jüngste Eintrag dieses Wertes aus dem Stack entfernt und wieder oben auf den Stack gelegt und die Methode liefert den Wert **`true`**.

Achtung: Beachten Sie die Zugriffsrechte der Attribute der Klasse **`Stack`**! Die Klasse **`Stack`** darf nicht verändert werden!

Tipp: Nutzen Sie zur Realisierung der Methode **`toTop`** ein Objekt der Klasse **`Stack`** als Hilfsobjekt. Der betroffene Stack wird Wert für Wert abgetragen (geerbte Methode **`pop`**) und im Hilfsstack gespeichert bis der gesuchte Wert ggfls. gefunden wird. Anschließend wird mit den im Hilfsstack gespeicherten Werten der betroffene Stack wieder gefüllt (geerbte Methode **`push`**) und dann der ggfls. entfernte Wert oben drauf gelegt.

Beispiel: Das folgende Testprogramm sollte die Ausgabe **`6 2 5 4 3 2 1`** erzeugen:

```
ToTopStack stack = new ToTopStack(50);
stack.push(1);
stack.push(2);
stack.push(3);
stack.push(2); // (*) jüngster Wert 2 im Stack
stack.push(4);
stack.push(5);
stack.toTop(2); // der bei (*) eingetragene Wert ist gemeint
stack.push(6);
while (!stack.isEmpty()) {
  System.out.print(stack.pop() + " ");
}
```

Aufgabe 20:

Definieren Sie eine Klasse `Texteditor`. Ein Texteditor repräsentiert dabei eine Folge von Zeichen (char) sowie einen Cursor, d.h. eine Position, an einer Stelle vor, zwischen oder hinter den Zeichen.

Die Klasse soll folgende Methoden definieren:

- Einen Konstruktor, der eine leere Zeichenkette enthält
- Eine Methode void delete(), mit der das Zeichen links vom Cursor gelöscht wird
- Eine Methode void type(char ch), bei der das übergebene char-Zeichen links vom Cursor in die Zeichenfolge eingefügt wird
- Eine Methode void left(), die den Cursor um eine Position nach links verschiebt
- Eine Methode void right(), die den Cursor um eine Position nach rechts verschiebt
- Eine Methode String getTextAndClear(), die die aktuelle Folge von Zeichen als String liefert und den Texteditor leert.

Operationen, die zu Fehlern führen können, sollen nicht ausgeführt werden (bspw. wenn sich der Cursor beim Löschen bereits ganz links befindet)

Für die Implementierung gilt folgende Einschränkung:
Nutzen Sie als interne Datenstruktur der Klasse Texteditor ausschließlich zwei Attribute vom Typ einer Klasse `Stack`. Diese ist analog zu der aus der Vorlesung bekannten Klasse Stack, speichert allerdings anstelle von String-Objekten char-Werte. Auf dem einen Stack sollen dabei alle Zeichen des Texteditor gespeichert

werden, die links vom Cursor stehen, auf dem anderen Stack alle Zeichen, die rechts vom Cursor stehen.

Test: das folgende Programm sollte „hello world!“ auf die Konsole ausgeben:

```
	public static void main(String[] args) {
		TextEditor editor = new TextEditor();
		editor.type('h');
		editor.type('e');
		editor.type('w');
		editor.type('o');
		editor.left();
		editor.left();
		editor.type('l');
		editor.type('l');
		editor.type('o');
		editor.type(' ');
		editor.right();
		editor.right();
		editor.type('r');
		editor.type('l');
		editor.type('d');
		editor.type('!');
		System.out.println(editor.getTextAndClear());
	}
```

Aufgabe 21:

Vorgegeben sind folgende Klassen **`Element`** und **`List`**. Sie implementieren die Datenstruktur einer „verketteten Liste“. Eine verkettete Liste ist eine dynamische Datenstruktur, die eine Speicherung von miteinander in Beziehung stehenden Objekten erlaubt. Die Anzahl der Objekte ist im Vorhinein nicht bestimmt. Die Liste wird durch Referenzen auf die jeweils folgende(n) Elemente realisiert.

```
class Element {
	int value; // Speicher fuer einen Wert
	Element next; // Referenz auf das folgende Element (null fuer Ende)

	Element(int v, Element n) {
		this.value = v;  this.next = n;
	}
}

class List {
	Element first; // erstes Element der Liste
	Element last; // letztes Element der Liste

	List() {
		this.first = null; this.last = null;
	}

	void append(int i) {
		Element elem = new Element(i, null);
		if (this.first == null) { // erstes Element in der Liste
			this.first = elem;
			this.last = elem;
```

```
            } else { // Anhaengen des Elementes am Ende
                  this.last.next = elem;
                  this.last = elem;
            }
      }

      void print() {
            Element elem = this.first;
            while (elem != null) {
                  System.out.print(elem.value);
                  if (elem.next != null) {
                        System.out.print(" -> ");
                  }
                  elem = elem.next;
            }
            System.out.println();
      }
}
```

Leiten Sie von der oben definierten Klasse **`List`** eine Klasse **`ExtList`** ab, die eine zusätzliche Methode **`void addPrev(int v)`** definiert. Ein Aufruf der Methode soll bewirken, dass die Elemente der Liste mit dem Wert des aktuellen Parameters v gesucht werden und zu ihren Vorgängerelementen (falls vorhanden) der Wert von v addiert wird.

Beispiel: Die Liste

`4 -> 2 -> 3 -> 4 -> 2 -> 4 -> 4`

wird durch den Aufruf der Methode **`addPrev(4)`** zu der Liste

`4 -> 2 -> 7 -> 4 -> 6 -> 8 -> 4`