Programmierkurs Java

Dr. Dietrich Boles

Aufgaben zu UE27-Vererbung
(Stand 13.04.2012)

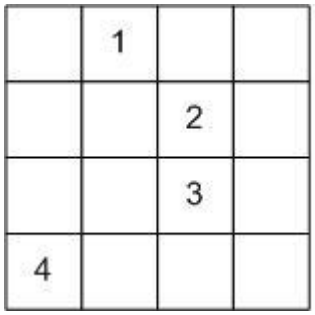
Aufgabe 1:

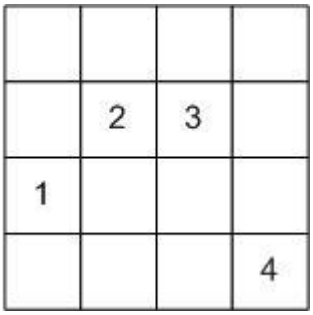
Stellen Sie sich vor, jemand hat ein bestimmtes Spiel implementiert. Dieses besteht u. a. aus folgenden Klassen:

```
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 unrelevante 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` natürlich nicht verändern. Achten Sie darauf, dass das intern durch die Matrix repräsentierte Spielfeld als `private` deklariert ist.





Spielfeld um 90 Grad nach links gedrehtes Spielfeld

Aufgabe 2:

Stellen Sie sich vor, Sie benötigen eine Klasse, die eine Menge 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;
      }
}
```

Implementieren Sie die Klasse *Menge*, indem Sie die Klasse *Container* nutzen!

Aufgabe 3:

Sie haben folgende Klasse `Stack` zur Verfügung:

```
class Stack {

      String[] store; // zum Speichern von Daten

      int current; // aktueller Index

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

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

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

```
    void push(String value) {
        if (!this.isFull())
            this.store[++this.current] = value;
    }

    String pop() {
        if (!this.isEmpty())
            return this.store[this.current--];
        return null; // Fehler!
    }
}
```

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.

1. 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.
2. Schreiben Sie weiterhin ein kleines Programm zum Testen der Klasse `ResortStack`.

Aufgabe 4:

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!

Aufgabe 5:

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 6:

Die folgenden Klassen implementieren die Ihnen aus der Vorlesung bekannte Datenstruktur „verkettete Liste“:

```
class Element {
      int value;

      Element next;

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

```
class List {
     Element first, last;

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

     void append(int i) { // Anhaengen am Ende
          Element elem = new Element(i, null);
          if (this.first == null) {
               this.first = elem;
               this.last = elem;
          } else {
               this.last.next = elem;
               this.last = elem;
          }
     }
}
```

(a) Leiten Sie von der Klasse *List* eine Klasse ab und überschreiben Sie die Methode `append` derart, dass kein `int`-Wert mehrfach in die Liste eingetragen werden kann.

(b) Definieren Sie eine zusätzliche Methode void prepend(int i), mit der ein int-Wert vorne in die Liste eingetragen werden kann.

(c) Definieren Sie weiterhin eine zusätzliche Methode `void change(int i)`, 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:

Die folgenden Klassen implementieren die Ihnen aus der Vorlesung (UE 25) bekannte Datenstruktur „verkettete Liste“:

```
class Element {
     int value;

     Element next;

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

class List {
     Element first, last;

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

     void append(int i) { // Anhaengen am Ende
          Element elem = new Element(i, null);
          if (this.first == null) {
               this.first = elem;
               this.last = elem;
          } else {
               this.last.next = elem;
               this.last = elem;
          }
     }
}
```

Leiten Sie von der Klasse *List* eine Klasse 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:

Sie haben folgende Klasse `Stack` zur Verfügung:

```
class Stack {

	String[] store; // zum Speichern von Daten

	int current; // aktueller Index

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

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

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

	void push(String value) {
		if (!this.isFull())
			this.store[++this.current] = value;
	}

	String pop() {
		if (!this.isEmpty())
			return this.store[this.current--];
		return null; // Fehler!
	}
}
```

Sie benötigen jedoch für eine bestimmte Anwendung einen Stack, bei dem zu bestimmten Zeitpunkten der Stack sortiert werden soll

1. 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.
2. Schreiben Sie weiterhin ein kleines Programm zum Testen der Klasse `SortStack`.

Aufgabe 9:

Die folgenden Klassen implementieren die Ihnen aus der Vorlesung bekannte Datenstruktur „verkettete Liste“:

```
class Element {
	int wert;
```

```
	Element naechstes;

	Element(int v, Element n) {
		this.wert = v;
		this.naechstes = n;
	}
}

class Liste {
	Element erstes, letztes;

	Liste() {
		this.erstes = null;
		this.letztes = null;
	}

	void hinzufuegen(int i) { // Anhaengen am Ende
		Element elem = new Element(i, null);
		if (this.erstes == null) {
			this.erstes = elem;
			this.letztes = elem;
		} else {
			this.letztes.naechstes = elem;
			this.letztes = elem;
		}
	}
}
```

Leiten Sie von der Klasse `Liste` eine Klasse ab, die eine zusätzliche Methode `void ansEndeSchieben(int i)` definiert. Ein Aufruf der Methode soll bewirken, dass alle Elemente der Liste mit dem Wert des aktuellen Parameters `i` ans 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 10:
Die folgenden Klassen implementieren die Ihnen aus der Vorlesung bekannte Datenstruktur „verkettete Liste“:

```
class Element {
	int wert;

	Element naechstes;

	Element(int v, Element n) {
		this.wert = v;
		this.naechstes = n;
	}
}

class Liste {
	Element erstes, letztes;

	Liste() {
		this.erstes = null;
		this.letztes = null;
	}

	void hinzufuegen(int i) { // Anhaengen am Ende
		Element elem = new Element(i, null);
		if (this.erstes == null) {
			this.erstes = elem;
			this.letztes = elem;
		} else {
			this.letztes.naechstes = elem;
			this.letztes = elem;
		}
	}
}
```

Leiten Sie von der Klasse `Liste` eine Klasse `EListe` ab, die eine zusätzliche Methode `void verdoppeln(int i)` definiert. Ein Aufruf der Methode soll bewirken, dass alle Elemente der Liste mit dem Wert des aktuellen Parameters `i` verdoppelt werden. Die Klassen `Element` und `Liste` dürfen nicht verändert 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:

Ihnen steht folgende Klasse `Stack` zur Verfügung:

```
class Stack {

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

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

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

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

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

	String pop() {
		if (!this.isEmpty()) {
			return this.store[this.current--];
		}
		return null; // Fehler!
	}
}
```

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:

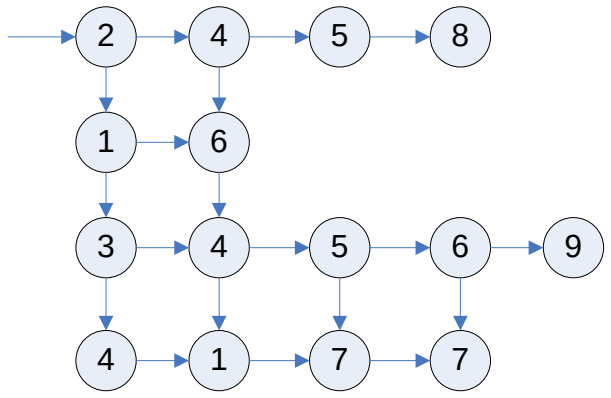
```
CountStack stack = new CountStack(4);
while (!stack.isFull()) {
      stack.push(IO.readString("Eingabe: "));
}
stack.push("noch einer");
while (!stack.isEmpty()) {
      System.out.println(stack.pop());
}
while (!stack.isFull()) {
      stack.push(IO.readString("Eingabe: "));
}
System.out.println(stack.getNumberOfValidPushOperations());

// Ausgabe sollte 8 sein!
```

Aufgabe 11:

Sie haben in der Vorlesung die Datenstruktur der „verketteten Liste“ kennen gelernt. 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 OutOfBoundsException extends Exception { }

// 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
	 * @throws OutOfBoundsException
	 *             wird bei fehlerhaften reihe- bzw. spalte-Indizes
geworfen
	 */
	public int getWert(int reihe, int spalte) throws OutOfBoundsException
{	}

	/**
	 * 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

	 * @throws OutOfBoundsException
	 *             wird bei fehlerhaften reihe- bzw. spalte-Indizes
geworfen
	 */
	public void setWert(int wert, int reihe, int spalte)
	    throws OutOfBoundsException {
	   // 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
```