VL04, Lösung 1

In der Vorlesung wurden die Klassen `LinkedList<E>` und `ArrayList<E>` vorgestellt. Die beiden Klassen implementieren das Interface `List<E>`. Die wichtigsten Methoden, die sie gemeinsam haben, sind:

```
size
add
remove
contains
iterator
get
set
```

Unterschiede bestehen in der Implementierung:

- Die Klasse `LinkedList` realisiert eine doppelt verkettete lineare Liste.
- Die Klasse `ArrayList` realisiert eine lineare Liste als dynamisches Array.

Unterschiede bestehen aufgrund der unterschiedlichen Implementierungen im Zeitverhalten:

- Die Einfüge- und Löschoperationen sind bei einer `LinkedList` performanter.
- Der wahlfreie Zugriff über einen Index ist bei der `ArrayList` performanter.

Weitere Unterschiede bestehen im Speicherbedarf:

- Eine `LinkedList` benötigt nur den Speicher für die tatsächlich vorhandenen Elemente.
- Eine `LinkedList` benötigt jedoch bei jedem `Link`-Element zusätzlich zu dem Speicherplatz für die Referenz auf die Nutzdaten Speicherplatz für die Zeiger auf den jeweiligen Vorgänger und Nachfolger.
- Eine `ArrayList` benötigt Speicherplatz nur zum Speichern sämtlicher Zeiger auf die Nutzdaten. Beim Anlegen eines `ArrayList`-Objekts wird immer bereits eine Minimalgröße für das interne Array reserviert. Zudem wird die einmal erreichte Größe nicht automatisch reduziert, wenn Elemente gelöscht werden.

VL04, Lösung 2

Fragestellung	Empfehlung und Begründung
Es soll eine Liste erzeugt werden, die allgemein aus sehr wenigen Elementen besteht, zwischenzeitlich aber auch aus sehr vielen Elementen.	**LinkedList** Es wird jederzeit nur der tatsächliche Speicherplatz benötigt. Aber: welcher Zugriff ist häufig?
Sie benötigen einen Stack (Stapel).	**LinkedList oder ArrayList** Die doppelte Verkettung erlaubt bei einer `LinkedList` den einfachen Zugriff auch auf den Vorgänger. Bei einer `ArrayList` ist der schnelle Zugriff auf den Vorgänger durch den Indexzugriff abgedeckt, allerdings ist die Größe des Stacks statisch. Anmerkung: `LinkedList` implementiert das Interface `Deque`, das bereits Stack-Methoden (`push`, `pop`, `peek`) anbietet. Eine `LinkedList` kann somit direkt als Stack verwendet werden.
Sie benötigen eine Queue (Warteschlange).	**LinkedList** Bei einer `ArrayList` müssen nach jedem Löschen alle Einträge umkopiert werden. Eine `LinkedList` hat allgemein Vorteile bei häufigem Hinzufügen oder Entfernen von Elementen. Anmerkung: `LinkedList` implementiert das Interface `Queue`, das bereits Methoden für Queues anbietet (`add`, `remove`, `element`). Eine `LinkedList` kann also direkt als Warteschlange verwendet werden. Anmerkung: ein einfaches Array kann eine Warteschlange sehr effizient als Ringpuffer implementieren, ohne dass Elemente beim Einfügen oder Löschen umkopiert werden müssen.
Sie benötigen eine Liste aus fortlaufend nummerierten Einträgen, wobei Sie überwiegend über den Index auf einzelne Elemente zugreifen.	**ArrayList** Schneller wahlfreier Zugriff

Fragestellung	Empfehlung und Begründung
Sie müssen stets mit einem möglichst geringen Speicherplatz auskommen.	**LinkedList** Nur der tatsächlich benötigte Speicherplatz wird belegt.

VL04, Lösung 3

```
import java.util.*;

public class ListInterfaceAufgabe
{
    // Elemente in Liste einfügen
    static void fillList(List<String> list)
    {
        // Zahlen von 0 bis 20 als Zeichenketten (Strings) einfügen
        for (int a = 0; a <= 20; a++)
            list.add("" + a);

        // Element an der Position 3 entfernen
        list.remove(3);

        // Erstes Element in der Liste entfernen, das gleich "6" ist
        list.remove("6");
    }

    // Liste vom Anfang bis zum Ende mit einer
    // foreach-Schleife iterieren und Elemente ausgeben
    static void printList(List<String> list)
    {
        for (String st : list)
            System.out.print(st + " ");

        System.out.println("\n-");
    }

    // Alle Elemente aus der Liste entfernen, die durch 5 teilbar sind
    static void remove5List(List<String> list)
    {
        ListIterator<String> it = list.listIterator();

        while (it.hasNext())
        {
            String st = it.next();

            // Eine Zahl ist durch 5 teilbar, wenn Sie ungleich 0 ist und
            // mit der Ziffer 0 oder 5 endet
            if ((st.endsWith("0") || st.endsWith("5")) && !st.equals("0"))
                it.remove();
        }
    }
```

```
    public static void main(String[] args)
    {
        // Erzeugen der LinkedList
        LinkedList<String> list1 = new LinkedList<String>();
        fillList(list1);
        System.out.println("\nAusgabe der ersten Liste(list1)");
        printList(list1);

        remove5List(list1);
        System.out.println("\nlist1 nach dem Entfernen der durch 5 " +
            "teilbaren Zahlen");
        printList(list1);

        // Erzeugen der ArrayList
        ArrayList<String> list2 = new ArrayList<String>();
        fillList(list2);

        System.out.println("\nAusgabe der zweiten Liste(list2)");
        printList(list2);

        // Teilliste von list2 erzeugen, die mit "7" beginnt
        // und mit "13" endet
        // Die Lösung setzt voraus, dass die Zahlen in der
        // Liste sortiert vorliegen.
        int anfang = list2.indexOf("7");

        // Endindex muss um 1 erhöht werden, da das Element mit
        // dem Endindex nicht mehr in die Teilliste aufgenommen
        // wird (siehe Beschreibung der Operation sublist)
        int ende = list2.indexOf("13")+1;

        List<String> teillist = list2.subList(anfang, ende);
        System.out.println("\nAusgabe der Teilliste");
        printList(teillist);

        teillist.remove("11");
        System.out.println("\nAusgabe der Teilliste nach Löschen der 11");
        printList(teillist);

        System.out.println("\nErneute Ausgabe von list2");
        printList(list2);
        // Entfernen von "11" aus der Teilliste führt dazu, dass
        // "11" auch aus der Gesamtliste list2 entfernt wird!
    }
}
```

VL04, Lösung 4

```
public class Stack<E> implements StackI<E>
{
    // Array, in dem die Elemente des Stacks gespeichert werden.
    // Das oberes Ende des Stacks liegt an Position pos-1.
    // Ein Array mit Elementen vom Typ E kann zwar deklariert, aber
    // nicht über new erzugt werden (Java-Mangel)!
    private Object[] st;

    // Nächste freie Position im Array
    // Gleichzeitig Anzahl der im Array/Stack gespeicherten Elemente
    private int pos;

    // Erzeugt ein Stack-Objekt, in dem maximal size Elemente
    // abgespeichert werden können
    public Stack(int size)
    {
        st = new Object[size];
    }

    // Legt übergebenes Element auf den Stack, sofern noch Platz
    // vorhanen ist. Das Element wird an Position pos gespeichert.
    public void push(E element)
    {
        if (pos < st.length)
            st[pos++] = element;
    }

    // Holt oberstes Element vom Stack, sofern der Stack nicht leer ist.
    public void pop()
    {
        if (pos > 0)
            pos--;
    }

    // Gibt oberstes Element auf dem Stack zurück, sofern der Stack nicht
    // leer ist. Bei leerem Stack wird null zurückgegeben.
    public E top()
    {
        return (pos > 0) ? (E)st[pos-1] : null;
    }

    // Gibt true zurück, falls der Stack leer ist
    public boolean isEmpty()
    {
        return pos <= 0;
    }
}
```