Einführung zur Aufgabengruppe 1

Alle Literaturhinweise beziehen sich auf die Vorlesung
Modellierung und Programmierung 1

1 Dynamische Verwaltung großer Datenmengen

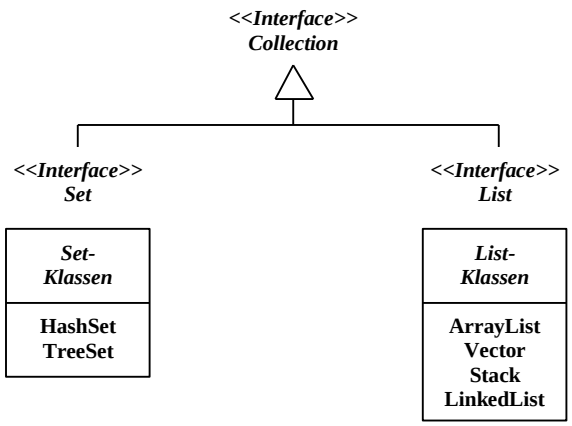
1.1 Collection-Klassen `java.util.*` (s. Kapitel Collection)

Felder

Gewöhnliche Felder sind Referenzdatentypen. Diese werden *statisch* angelegt und immer dann verwendet, wenn man *im Voraus* bereits die Anzahl der Feldkomponenten kennt. Die Feldkomponenten können sowohl Elementar- als auch Referenzdatentypen aufnehmen. Überschreitet man die *Feldgrenzen*, kommt es zu einem *Laufzeitfehler*. Falls man diesen nicht auffängt, bricht das Programm ab (`ArrayIndexOutOfBoundsException`).

Collection-Klassen

Collection-Klassen bieten bei der Verwaltung großer Datenmengen mehr Möglichkeiten als gewöhnliche Felder. Die Größe einer *Collection* kann jeder Zeit *dynamisch* an die gewünschte Anzahl der Elemente angepasst werden.
Das Interface `Collection<E>` definiert eine Schablone (**Template**) für Collections verschiedener Datentypen, `E` ist der Typparameter.



1.1.1 Interface `Set<E>`

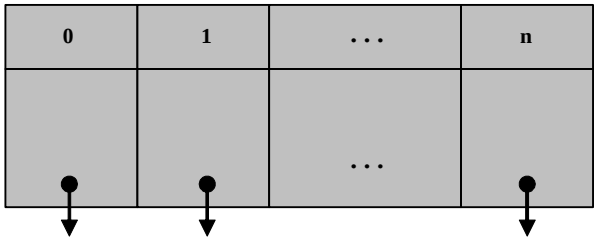
Mengenoperationen werden durch entsprechende Methoden realisiert, **`n`** und **`m`** seien Objekte einer Set-Klasse:

M ∪ N	≙	**`m.addAll( n)`**
M ∩ N	≙	**`m.retainAll( n)`**
M \ N	≙	**`m.removeAll( n)`**

Klasse `HashSet<E>` — ***verwaltet Mengen, ungeordnet***
Klasse `TreeSet<E>` — ***verwaltet Mengen, geordnet***

1.1.2 Interface `List<E>`

Klassen `ArrayList<E>`, `Vector<E>` ***verwalten lineare Listen als Feld***



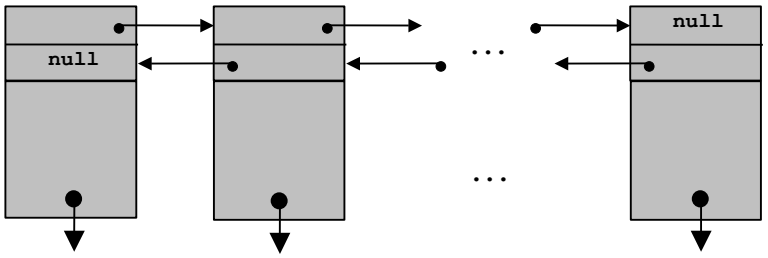
Index

Referenzen auf die Komponenten der Liste

Klasse `Stack<E>` ***verwaltet Keller***

Die Klasse **`Stack<E>`** ist eine Ableitung der Klasse **`Vector<E>`**.

Klasse `LinkedList<E>` ***verwaltet doppelt verkettete lineare Listen***



Verweis auf Nachfolger

Verweis auf Vorgänger

Referenzen auf die Komponenten der Liste

1.1.3 Klassen `Arrays` und `Collections`

Diese Klassen fassen Klassenmethoden zum *Suchen* und *Sortieren* in *gewöhnliche Felder* bzw. *Collection-Klassen* zusammen, wobei die Klasse **`Arrays`** für *einfache Datentypen* und die Klasse **`Collections`** für *Referenzdatentypen* zuständig sind.

1.2 Zusammenfassung

- ⇨ Ist die Komponentenanzahl eines Feldes bekannt, so verwendet man *statische* Felder.
- ⇨ Bleibt die Liste *klein*, wird hauptsächlich *wahlfrei* darauf zugegriffen, überwiegen die *lesenden* gegenüber den schreibenden Zugriffen deutlich, so liefert die **`ArrayList<E>`** die besten Ergebnisse.
- ⇨ Ist die Liste dagegen sehr *groß* und werden *häufig* Einfügungen und Löschungen vorgenommen, ist die **`LinkedList<E>`** die bessere Wahl.
- ⇨ Wird von *mehreren* Anwendungen *gleichzeitig* auf die Liste zugegriffen, sollte die Klasse **`Vector<E>`** verwendet werden, denn ihre Methoden sind bereits weitgehend synchronisiert.
- ⇨ Sollen *Duplikate* der Werte nicht abgespeichert werden, so verwendet man keine *List-Klasse*, sondern eine der *Set-Klassen*.

1.3 Datenzugriff

Collection	*Eingaben*		*Ausgaben*				
	add	**push**	**for(E obj:col)**	**Iterator**	**get**	**ListIterator**	**pop**
Set							
HashSet	**x**		**x**	**x**			
TreeSet	**x**		**x**	**x**			
List							
ArrayList	**x**		**x**	**x**	**x**	**x**	
Vector	**x**		**x**	**x**	**x**	**x**	
Stack	**x**	**x**	**x**	**x**	**x**	**x**	**x**
LinkedList	**x**		**x**	**x**	**x**	**x**	

```
Collection<E> col;
  col.add( E obj);                                                    // Eingabe

  for( E obj: col) System.out.print( " " + obj)                       // Ausgaben
  Iterator<E> it = col.iterator(); while( it.hasNext()) System.out.print( " " + it.next());

List<E> list;
  for( int i = 0; i < list.size(); i++) System.out.print( " " + list.get( i));      // Ausgaben
  ListIterator<E> itNf = list.listIterator( 0);
      while( itNf.hasNext()) System.out.print( " " + itNf.next());
  ListIterator<E> itVg = list.listIterator(list.size());
      while( itVg.hasPrevious()) System.out.print( " " + itVg.previous());

Stack<E> stack;                                                       // Ein- und Ausgabe
  stack.push( E obj);
  while(true){ try{ System.out.print( " "+stack.pop());} catch(EmptyStackException e){ break;}}
```

2 Beispiel „Spiegelzahlen“

2.1 Aufgabenstellung

Spiegelzahlen

Erzeugen der zu einer beliebig langen natürlichen Zahl gehörigen symmetrischen Zahl mittels Spiegelzahlen:

Diejenige natürliche Zahl, die zu einer vorgegebenen Zahl die spiegelverkehrte Zifferndarstellung besitzt, bezeichnet man als ihre **Spiegelzahl**. Als **symmetrisch,** oder auch *Palindrom*, bezeichnet man eine Zahl, die mit ihrer Spiegelzahl übereinstimmt.

Addiert man eine natürliche Zahl mit ihrer Spiegelzahl, ist das Ergebnis im Allgemeinen *nicht* symmetrisch. Verfährt man mit dem Ergebnis ebenso, erhält man nach endlich vielen Schritten eine symmetrische Zahl.

Ein allgemeingültiger Beweis dieser Vermutung steht bis heute aus. Bei verschiedenen Zahlen, wie zum Beispiel 196, bricht der Algorithmus wahrscheinlich niemals ab.

Beispiel

Zahl	**53978**	
gespiegelt	+87935	
addiert	141913	
gespiegelt	+319141	
addiert	461054	
gespiegelt	+450164	
addiert	911218	
gespiegelt	+812119	
addiert	1723337	
gespiegelt	+7333271	
addiert	9056608	
gespiegelt	+8066509	
addiert	17123117	
gespiegelt	+71132171	
addiert	**88255288**	7 Schritte

Aufgabenstellung

Entwickeln Sie ein System von Klassen, welches beliebig lange natürliche Zahlen verwaltet, auf Symmetrie überprüft und mit dem Spiegel der Zahl addiert. Erzeugen Sie durch wiederholte Spiegelung symmetrische Zahlen.

Testprogramm

Geben Sie die folgenden Zahlen und Schrittweiten (Anzahl der Spiegelungen bis zur Ausgabe eines Zwischenergebnisses) ein, überprüfen Sie die Ergebnisse.

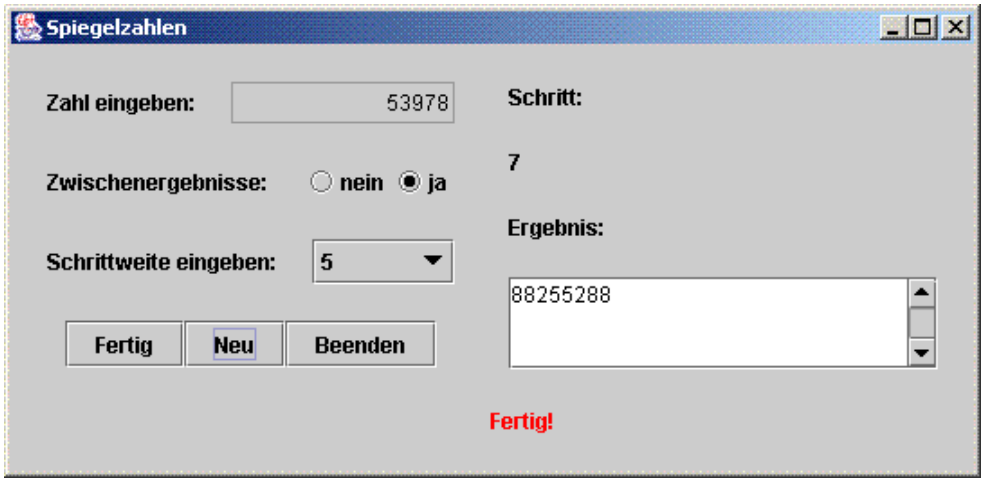
Zahl	Schrittweite	Schrittanzahl	Wert
789	1	4	66066
53978	2	7	88255288
52007615407	10	8	89887611678898
456789900345345	1000	46000	kein Ergebnis!

Eingabe
Natürliche Zahl beliebiger Länge, Schrittweite.
Ausgabe
Schrittanzahl, Zwischenergebnis bzw. symmetrische Zahl.
Abbruch
Das Programm bricht nach Ausführung ab.

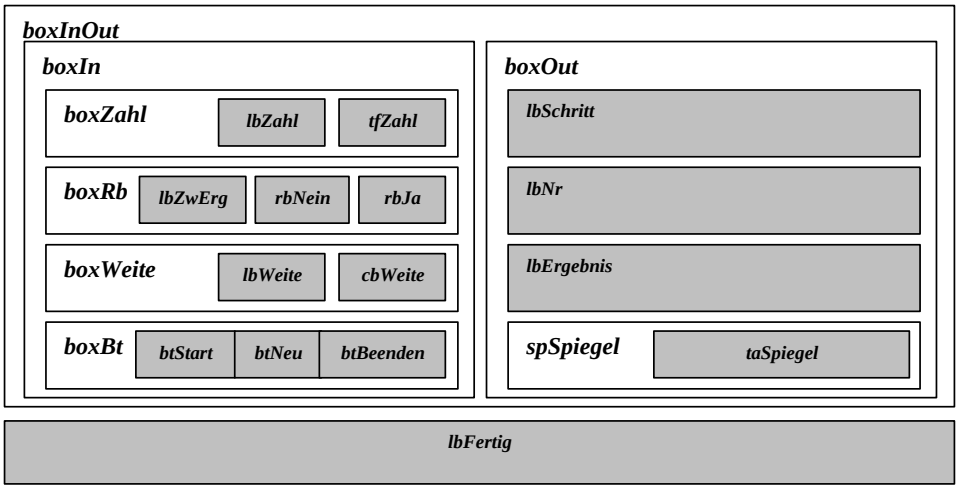
Bemerkung
Spiegelzahlen lassen sich auch von der Klasse **`BigInteger`**, einer Klasse für beliebig lange ganze Zahlen, ableiten. Wir wollen diese Aufgabe aber mittels *Collection-Klassen* lösen.

Modellierung und Programmierung **Klassendiagramm, Dokumentation, *SpiegelZahl.zip***

2.2 Zusatz: Benutzerschnittstelle zu Spiegelzahlen



Aufteilung des Fensters unter Verwendung von Boxen als Container (`BoxLayout`):



Modellierung und Programmierung mit MVC-Architektur
Klassendiagramm, Dokumentation, *SpiegelZahlMVC.zip*