Sven Eric Panitz

13. Januar 2023

Inhaltsverzeichnis

Array-basierte iterative Listen

1 Einführung

In dieser Aufgabe geht es darum, eine Listenklasse mit einer Reihung als Datenhaltung umzusetzen.

Hierzu werden wir zunächst eine Listenschnittstelle definieren, die generisch über den Elementtyp der Liste ist. Diese Schnittstelle wird dann implementiert über eine Klasse, die eine Reihung zur Speicherung der Listenelemente enthält.

In einem dritten Schritt werden wir über Standardmethoden in der Schnittstelle weitere Listenfunktionalität umsetzen.

Damit erfüllt dieses Übungsaufgabe zwei primäre Lernzziele:

- Arbeiten mit generischen Klassen und Schnittstellen
- Abstraktion der Funktionalität mit Hilfe von Schnittstellen

List.java

```
package name.panitz.util;
```

2 Die Listen-Schnittstelle

Die Listenschnittstelle ist generisch über den Typ der Elemente. Daher führen wir die Typvariable `E` ein.

List.java

```
public interface List<E>{
```

Damit muss für jedes konkretes Listenobjekt entschieden werden, welchen Typ `E` annehmen soll. Der Typ ist dann beliebig aber fest.

2.1 Abstrakte Methoden der Schnittstelle

Eine Liste ist eine Sammlung von Objekten. Die Objekte sind in einer festen Reihenfolge und sind anfangend von 0 indiziert.

Die ersten wichtigste Funktion ist daher, die Anzahl der Elemente einer Liste zu erfragen.

List.java

```
  int size();
```

Desweiteren muss es möglich sein, das Element an einem bestimmten Index zu selektieren.

List.java

```
  E get(int i);
```

Jede Liste soll eine Methode haben, mit der ein weiteres Element am Ende der Liste hinzugefügt werden kann.

List.java

```
  void add(E el);
```

3 Die Implementierung als array-basierte Liste

Die eigentliche Implementierung ist die generische Klasse `AL`. Sie ist generisch gehalten über die Elemente der Liste. `AL` steht dabei für Array List.

List.java

```
class AL<E> implements List<E>  {
```

Intern benötigt sie zwei Felder. Eine Zahl, die anzeigt, wie viele Elemente in der Liste enthalten sind, und die eigentliche Reihung, in der die Elemente gespeichert sind.

List.java

```
  private int theSize = 0;
  private Object[] store = new Object[10];
```

Man mache sich klar, dass die Länge der Reihung `store.length` in der Regel größer ist als die Länge der Liste `size()`. Die Länge der Reihung stellt die derzeitige Kapazität der Liste dar, die `theSize` hingegen den tatsächlichen Füllgrad der Liste.

Somit lässt sich auch die Funktion `size` direkt über das Feld `theSize` ermitteln.

List.java

```
  @Override public int size(){
    return theSize;
  }
```

Ebenso hat man einen direkten Zugriff auf die Elemente über den Index.

List.java

```
  @SuppressWarnings("unchecked")
  public E get(int i){
    if (i>=size()||i<0) throw new IndexOutOfBoundsException();

    return (E)store[i];
  }
```

Hinzufügen von Elementen Das eigentliche Hinzufügen geschieht, indem am Index `theSize` das neue Element gespeichert wird und anschließend das Feld `theSize` um eins erhöht wird: `store[theSize++] = e`. Problematisch ist nur, wenn die Kapazität der Listenklasse ausgeschöpft ist, weil alle Indizes des Arrays bereits mit einem Element belegt sind. Dann sind die Elemente in einen neueren, größeren Array zu kopieren, der dann Platz für weitere Elemente hat.

List.java

```
  public void add(E e){
    if (theSize>=store.length) enlargeStore();
    store[theSize++] = e;
  }
```

Die Kopie der einzelnen Elemente in eine neue, größere Reihung ist in einer privaten Hilfsmethode ausgelagert.

List.java

```
private void enlargeStore(){
  Object[] newStore = new Object[store.length+10];
  for (int i=0;i<theSize;i++) newStore[i]=store[i];
  store=newStore;
}
```

Stringdarstellung Die Methode `toString` listet die Listenelemente in eckigen Klammern mit Komma separiert auf.

List.java

```
@Override public String toString(){
  StringBuffer result = new StringBuffer("[");
  for (var i=0;i<size();i++){
    if (i>0) result.append(", ");
    result.append(store[i]);
  }
  result.append("]");
  return result.toString();
}
```

Gleichheit Wir müssen explizit die Methode `equals` für unsere Listenklasse überschreiben.

List.java

```
@Override public boolean equals(Object o){
  if (o instanceof List that){
    if (this.size()!=that.size()) return false;
    for (int i=0;i<size();i++){
      if (!this.get(i).equals(that.get(i)))return false;
    }
    return true;
  }
  return false;
}
}
```

4 Statische und Standardmethoden der Listenschnittstelle

4.1 Erzeugen von Listen

Da wir keinen Konstruktor für die Klasse `AL` geschrieben haben, gibt es nur den leeren Konstruktor, der entsprechend eine leere Liste erzeugt.

Wir können diesen in einer statischen Methode der Schnittstellen kapseln.

List.java

```
static <E> List<E> of(){return new AL<>();}
```

4.2 Konstruktion mit variabler Argumentanzahl

Jetzt sehen wir eine statische Funktion vor, mit deren Hilfe eine Liste durch Aufzählung der Elemente erzeugt werden kann.

List.java

```
@SafeVarargs static <E> List<E> of(E...es){
  List<E> r = of();
  for (var e:es) r.add(e);
  return r;
}
```

4.3 Test auf leere Liste

Die elementare Testfunktion, ob die Liste überhaupt ein Elenemt enthält, lässt sich über das feld `size` direkt ausdrücken.

List.java

```
default boolean isEmpty(){return size()==0;}
```

4.4 Aufrufe auf der JShell

Einfache Testsaufrufe lassen sich auch für diese Listenimplementierung in der JShell durchführen.

Hierzu ist die Klasse mit dem Javacompiler zu übersetzen und die class-Dateien entsprechend der Paketstruktur in einem Ordner zu generieren:

Shell

```
javac --enable-preview -source 19 -d classes/ List.java
```

Beim Übersetzen ist mindestens die Java-Version 15 mit angeschalteten Preview zu verwenden. Die Option `-d classes` sorgt dafür, dass in dem Ordner `classes` die generierten Klassen entsprechend der Paketstruktur gespeichert werden.

Jetzt kann man eine JShell-Session öffnen:

Shell

```
jshell  --enable-preview --class-path classes
|  Welcome to JShell -- Version 19.0.1
|  For an introduction type: /help intro

jshell>
```

Hierbei ist der Ordner, in dem die generierten Klassen liegen, als Klassenpfad anzugeben.

Es empfiehlt sich, alle statischen Eigenschaften der Schnittstelle `List` in der JShell-Session zu importieren.

Shell

```
jshell> import static name.panitz.util.List.*;

jshell>
```

Jetzt kann interaktiv mit den Listen gearbeitet werden.

Shell

```
jshell> of()
$2 ==> []

jshell> of(1,2,3,4,5,6)
$3 ==> [1, 2, 3, 4, 5, 6]

jshell> of(1,2,3,4,5,6).get(2)
$4 ==> 3

jshell> of(1,2,3,4,5,6).isEmpty()
$5 ==> false

jshell> of("hallo","welt")
$6 ==> [hallo, welt]
```

4.5 Aufgaben

Es folgen viele kleine Aufgaben. Alle zu schreibenden Funktionen lassen sich am besten mit einer Schleife umsetzen.

Die Funktionen können sich natürlich gegenseitig aufrufen. Dieses ist vielfach gewünscht. Auch die Funktionen `size` und `get` können in dieser Umsetzung direkt verwendet werden.

Alle Funktionen können direkt in der JShell kurz getestet werden. Ein Beispielaufruf ist jeweils bei der Aufgabenstellung mit angegeben.

Aufgabe 1 Wir beginnen mit den vielen kleinen Listen verarbeitenden Funktionen:

a) Schreiben Sie die Funktion, die eine neue Teilliste erzeugt, die aus den ersten i-Elemente aus der this-Liste entsteht. Ist i größer als die Länge, dann sei das Ergebnis die komplette Liste. Ist i negativ, dann sei das Ergebnis die leere Liste.

List.java

```
default List<E> take(int i){
  List<E> rs = of();
  /*ToDo*/
  return rs;
}
```

b) Schreiben Sie die Funktion, die eine neue Liste erstellt, die mit den Elementen der this-Liste beginnt und anschließend die Elemente der that-Liste hat. Es soll also nicht die this-Liste verändert werden, sondern eine komplett neue Liste erzeugt werden.

List.java

```
default List<E> append(List<E> that){
  List<E> rs = of();
  return rs;  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).append(of(431,432,431,21,76))
$4 ==> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 431, 432, 431, 21, 76]
```

c) Schreiben eine Funktion, die eine Liste mit den Elementen in umgekehrter Reihenfolge erzeugt.

List.java

```
default List<E> reverse(){
  return of();   /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).reverse()
$19 ==> [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

d) Schreiben Sie jetzt eine Funktion, die testet, ob ein dem Argument gleiches Element in der Liste enthalten ist.

List.java

```
default boolean contains(E el) {
  return false;  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).contains(6)
$7 ==> true

jshell> of(1,2,3,4,5,6,7,8,9,10).contains(42)
$8 ==> false
```

e) Schreiben Sie eine Funktion, die das letzte Element der Liste zurück gibt. Ist die Liste leer, so soll eine `NoSuchElementException` geworfen werden.

List.java

```
default E last(){
  return null;   /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).last()
$13 ==> 10
```

f) Schreiben Sie jetzt die Funktion, die keine neue Liste erstellt, sondern die this-Liste verändert, indem die Elemente der that-Liste in ihr am Ende eingefügt werden.

List.java

```
default void addAll(List<E> that){
  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> var xs = of(1,2,3,4)
xs ==> [1, 2, 3, 4]

jshell> xs.add
add(      addAll(
jshell> xs.addAll(of(5,6,7,8,9))

jshell> xs
xs ==> [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

g) Schreiben Sie die Funktion, die eine Teilliste erzeugt, die ohne die ersten i-Elemente aus der this-Liste entsteht. Ist i größer als die Länge, dann sei das Ergebnis die leere Liste. Ist i negativ, so sei das Ergebnis die gesamte Liste.

List.java

```
default List<E> drop(int i){
  List<E> rs = of();
  return rs;  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).drop(6)
$9 ==> [7, 8, 9, 10]
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).take(6)
$10 ==> [1, 2, 3, 4, 5, 6]
```

h) Schreiben Sie die Funktion, die eine Teilliste erzeugt, die mit dem Element an dem angegebenen Index startet und die angegebene Länge hat. Sollten nicht genug Elemente in der Liste sein, so wird die maximale Anzahl der von dem Index an kommenden Elemente genommen.

List.java

```
default List<E> sublist(int from, int length) {
  return of();  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6,7,8,9,10).sublist(3,4)
$11 ==> [4, 5, 6, 7]

jshell> of(1,2,3,4,5,6).sublist(3,100000)
$2 ==> [4, 5, 6]

jshell> of(1,2,3,4,5,6).sublist(10,100000)
$3 ==> []
```

i) Schreiben Sie eine Funktion, die eine neue Liste erzeugt, in der zwischen den Elementen der this-Liste das übergebene Element steht.

List.java

```
default List<E> intersperse(E e){
  return of();      /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6).intersperse(42)
$2 ==> [1, 42, 2, 42, 3, 42, 4, 42, 5, 42, 6]
```

j) Schreiben Sie eine Funktion, die prüft, ob die this-Liste der Anfang der that-Liste ist.

List.java

```
default boolean isPrefixOf(List<E> that){
  return false;     /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4).isPrefixOf(of(1,2,3,4,5,6,7,8,9,10))
$2 ==> true
```

Beachten Sie, dass die leere Liste ein Präfix von jeder Liste ist.

k) Schreiben Sie eine Funktion, die prüft, ob die this-Liste das Ende der that-Liste ist.

List.java

```
default boolean isSuffixOf(List<E> that){
  return false;     /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(8,9,10).isSuffixOf(of(1,2,3,4,5,6,7,8,9,10))
$3 ==> true
```

l) Schreiben Sie eine Funktion, die prüft, ob die this-Liste in der that-Liste enthalten ist.

List.java

```
default boolean isInfixOf(List<E> that){
  return false;     /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(6,7,8,9).isInfixOf(of(1,2,3,4,5,6,7,8,9,10))
$4 ==> true
```

m) Schreiben Sie eine Funktion, die eine neue Liste erzeugt, bei der das ursprüngliche Kopfelement an die letzte Stelle wandert.

List.java

```
default List<E> rotate(){
  return of();  /*ToDo*/
}
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6).rotate()
$2 ==> [2, 3, 4, 5, 6, 1]
```

n) Schreiben Sie eine Funktion, die die Liste aller Listen erzeugt, mit der die this-Liste endet.

List.java

```
  default List<List<E>> tails(){
    return of(of());  /*ToDo*/
  }
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4).tails()
$4 ==> [[1, 2, 3, 4], [2, 3, 4], [3, 4], [4], []]
```

o) Schreiben Sie die Funktion, die die Elemente der this-Liste paarweise mit den Elementen der that-Liste zu einer Liste aus Paaren verknüpft.

Hierzu sei zunächst die Record-Klasse für Paare von Objekten definiert.

List.java

```
  record Pair<A,B>(A fst,B snd){
    @Override public String toString(){
      return "("+fst()+", "+snd()+")";
    }
  }
```

List.java

```
  default <B> List<Pair<E,B>> zip(List<B> that){
    return of();   /*ToDo*/
  }
```

Ein Beispielaufruf:

Shell

```
jshell> of(1,2,3,4,5,6).zip(of("A","B","C","D"))
$2 ==> [(1, A), (2, B), (3, C), (4, D)]
```

List.java

```
}
```