

# Dynamische Datenstrukturen II : Der Stapel

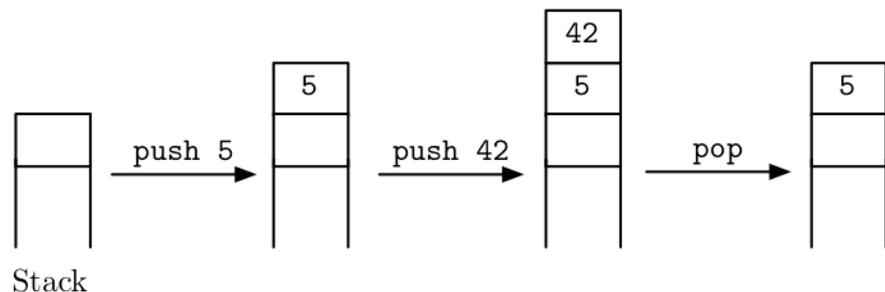
## Lernziele:

- Sie kennen die Datenstruktur Stack (Stapel) und können sie einsetzen
- Sie kennen wichtige Datenstrukturen, welche in der Collection-Library vorhanden sind
- Sie können das Interface *Iterator* für eine while-Schleife anwenden

## 1 Eine weitere Datenstruktur: Stapel (oder auch Stack)

Mit Hilfe der verketteten Liste, können wir auch eine weitere häufig benützte Datenstruktur umsetzen: den Stapel, bzw. Stack.

Der Stack arbeitet nach dem LIFO-Prinzip („last in – first out“). Als Analogie können Sie sich den Stack wie einen Stapel Teller in der Mensa vorstellen:



Die Grundoperationen für einen Stack sind:

- push* -> ein Element zuoberst hinzufügen  
*pop* -> das Element zuoberst entfernen

Die Datenstruktur, welche wir für die verkettete Liste erstellt haben (d.h. die Klasse *Node*) eignet sich sehr gut, um auch einen Stack umzusetzen.

Wir schreiben somit eine *Stack*-Klasse, welche die Datenstruktur der verketteten Liste verwendet (also wieder die Klasse *Node* verwenden):

```
public class MyStack {  
  
    //reference to the first element in stack (=last added element)  
    private Node first;  
    //counter  
    private int listCount;  
  
    //constructor  
    public MyStack(){  
        listCount = 0;  
    }  
  
    //add push and pop  
}
```

Wir fügen ein Element zuvorderst in die Kette, mit einer *push* Methode:

```
/**  
 * Adds element to the top of the list  
 * @param element  
 */  
public void push(Object value){  
  
    //the new element is the last in the list:  
    Node current = first;  
    first = new Node(value);  
    first.setNext(current);  
    listCount++;  
}
```

Und mit der Methode *pop* holen wir das zuoberste Element von der Liste:

```
/**  
 * Removes top element from stack  
 * @param element  
 */  
public Object pop(){  
  
    Object value = first.getItem();  
    first = first.getNext(); //now next element is the first  
    listCount--;  
    return value;  
}
```

Ein Aufruf in der main-Methode sieht somit wie folgt aus:

```
MyStack list = new MyStack();  
list.push(„to“);  
list.push(„be“);  
list.push(„or“);  
list.push(„not“);
```

## 2 Aufgaben

### 2.1 Stack schreiben

Implementieren Sie Ihre *Stack* Klasse mit den *push* und *pop* Methoden. Bei *push* wird ein Element zuoberst hinzugefügt. Bei *pop* wird das aktuelle Element entfernt. Verwenden Sie wiederum Ihre *Node* Klasse für die Datenstruktur.

### 2.2 Stack testen und Elemente ausgeben

Schreiben Sie eine Methode *toString()*, welche die Elemente vom Stack als String ausgibt. Mit dieser Methode können Sie auch Ihre *Stack*-Klasse testen. Sie beginnen mit dem ersten Element (d.h. das Element, welches zuoberst auf dem Stack ist). Ihre Iteration sieht so aus:

```
while (current != null){  
    //add value to string:  
    ...  
    //get next Node  
    current = current.getNext();  
}
```

### 2.2 isEmpty() Methode implementieren

Fügen Sie eine *isEmpty()* Methode hinzu, die überprüft, ob es Elemente im *Stack* hat. Die Methode gibt einen boolean zurück.

➔ Zeigen Sie Ihre Lösungen der Lehrperson.

### 2.3 Stack ergänzen

Ergänzen Sie *Stack* um eine Methode *peek()*, das das zuletzt auf dem Stapel gelegte Element zurückliefert (ohne es zu entfernen).

## 2.4 Palindrome prüfen

Verwenden Sie den *Stack* um zu überprüfen, ob ein String ein Palindrome ist (d.h. Wort oder Satz ist dasselbe von vorne und von hinten). Mit einem *Stack* lässt sich das einfach überprüfen. Erweitern Sie den Code, so dass der Benutzer einen String eingeben kann.

- Verwenden Sie für diese Lösung ihren eigenen *Stack* und die *Stack* Klasse der Java Collection.

## 3 Zusätzliche Aufgaben

### 3.1 Verwendung von *Iterator* statt normaler *for*-Schleufe

Anstatt eine übliche *for*-Schleufe zu implementieren, um alle Elemente im Stack anzuzeigen, wollen wir das Interface *Iterator* verwenden.

```
public class MyStack implements Iterator{

    //reference to the first element in stack (=last added element)
    private Node first;
    //counter
    private int listCount;

    ...
}
```

Das Interface *Iterator* verlangt zwingend dass wir folgende Methoden implementieren:

*hasNext()*           -> gibt true oder false zurück  
*next()*               -> gibt das nächste Element zurück  
*remove()*           -> entfernt ein Element

Uns interessiert vor allem *hasNext()* und *next()*. Für *hasNext()* prüfen wir, ob das Element zuoberst auf ein nächstes Element zeigt. Das sieht dann wie folgt aus:

```
@Override
public boolean hasNext() {
    return first != null; // return false if null
}
```

```
}
```

Wie muss die Methode *next()* aussehen, damit wir jeweils das nächste Element (und dessen Inhalt) bekommen? Ergänzen Sie den untenstehenden Code in Ihrem Programm:

```
@Override  
public Object next() {  
  
    Object item = first.getItem();  
    //TODO: complete code here  
    return item;  
}
```

Testen Sie diese Methoden, indem Sie eine while-Schleife für *hasNext()* implementieren.

→ Zeigen Sie Ihre Lösungen der Lehrperson.

### 3.2 Iteration mit Lambda

Seit Java 8 gibt es auch die sog. Lambda-Funktion, welche viele Such-, Iterations- und Replace-Methoden vereinfacht.

Wie sieht eine Iteration mit Lambda aus? Implementieren Sie diese Variante und vergleichen Sie diese zum Iterator.

## 4 Unsere Datenstrukturen und die Collections Bibliothek

Die bisherigen elementaren Datenstrukturen sind in Java auch in der Collections-Bibliothek aufgeführt. Somit finden wir für



```
while (i.hasNext()) {  
    String s = i.next();  
    System.out.println(s);  
}
```

Der *Iterator* selbst ist ein Interface. Sobald eine Klasse dieses Interface implementiert, hat es automatisch das Verhalten eines Iterators. Siehe dazu `java.util.Iterator`.

## 5 Aufgabe

### Array umgekehrt ausgeben mit Verwendung von *Iterator*

Schreiben Sie eine Klasse, welche ein Array entgegennimmt und die Elemente in umgekehrter Reihenfolge ausgibt. Implementieren Sie dabei das Interface *Iterator* und passen Sie die vom Interface verlangten Methoden an, also:

```
public boolean hasNext()
```

→ Prüfe, ob es noch Elemente im Array hat

```
public Object next()
```

→ Gib das Element in umgekehrter Position aus (also `data[--i]`)

Die Methode `remove()` lassen wir leer, weil wir es vermeiden wollen, Elemente aus einer Liste zu entfernen, welche wir gleichzeitig auch noch iterieren.

**Erweiterung:** Verwenden Sie die Lambda-Funktion statt *Iterator*.