

# Table of Contents

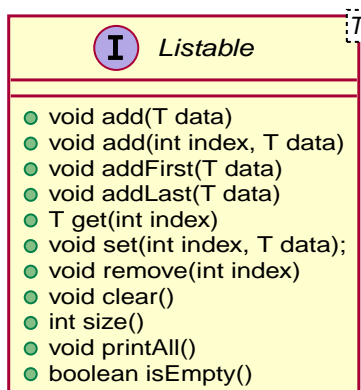
Appendix A: Leitfaden für die Listen-Implementierung .....	1
--	---

## Appendix A: Leitfaden für die Listen-Implementierung



Lesen bitte als Erstes die Folien zum Themengebiet Listen sowie diesen Leitfaden, bevor Sie mit der Bearbeitung und Umsetzung des Aufgabenblattes starten.

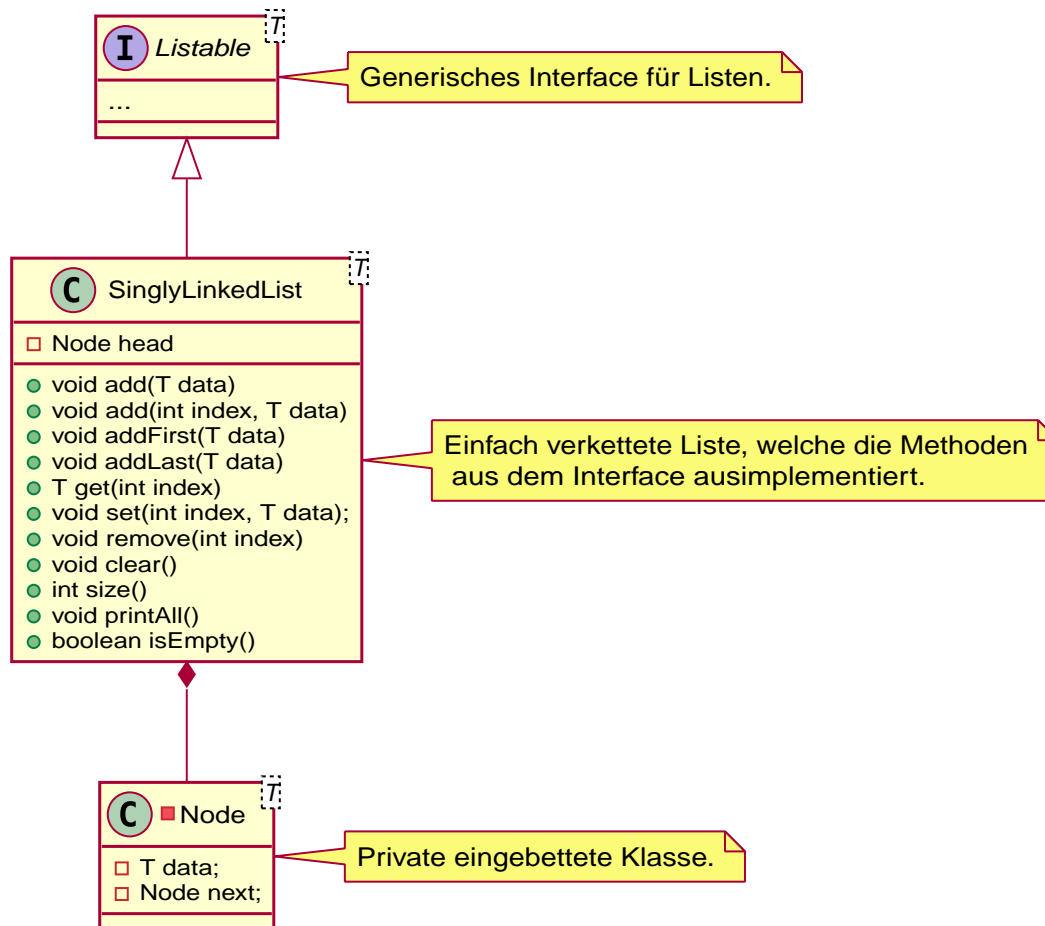
1. **Step** – Generisches Interface `Listable<T>` anlegen:



Bevor wir mit der Implementierung der Aufgabe starten, wird als Erstes ein **generisches Interface** `Listable<T>` angelegt. Dieses bildet die Grundlage für die Implementierung einer **einfach und doppelt verketteten Liste**. Deklarieren Sie das Interface wie folgt:

```
1 public interface Listable<T> {  
2  
3     void add(T data);  
4  
5     void add(int index, T data);  
6  
7     void addFirst(T data);  
8  
9     void addLast(T data);  
10  
11     void set(int index, T data);  
12  
13     T get(int index);  
14  
15     void remove(int index);  
16  
17     void clear();  
18  
19     int size();  
20  
21     void printAll();  
22  
23     boolean isEmpty();  
24 }
```

2. **Step** – Generische Klasse `SinglyLinkedList<T>` anlegen:



Als nächstes wird die Klasse **SinglyLinkedList** als **generische Klasse** angelegt.

```

1 public class SinglyLinkedList<T> implements Listable<T> {
2
3     [...]
4
5 }
  
```

Diese implementiert das Interface **Listable<T>** mit dem **Schlüsselwort** **implements**. Achten Sie besonders auf das **<T>**.



Nachdem Sie das Interface mit **Listable<T>** zur Klasse korrekt hinzugefügt haben, können Sie die Glühbirne oder das Kontextmenü verwenden, um alle **unimplemented Methods** mit nur einem Klick zu generieren. Diese Funktionalität gibt es in beiden IDEs Eclipse und IntelliJ.

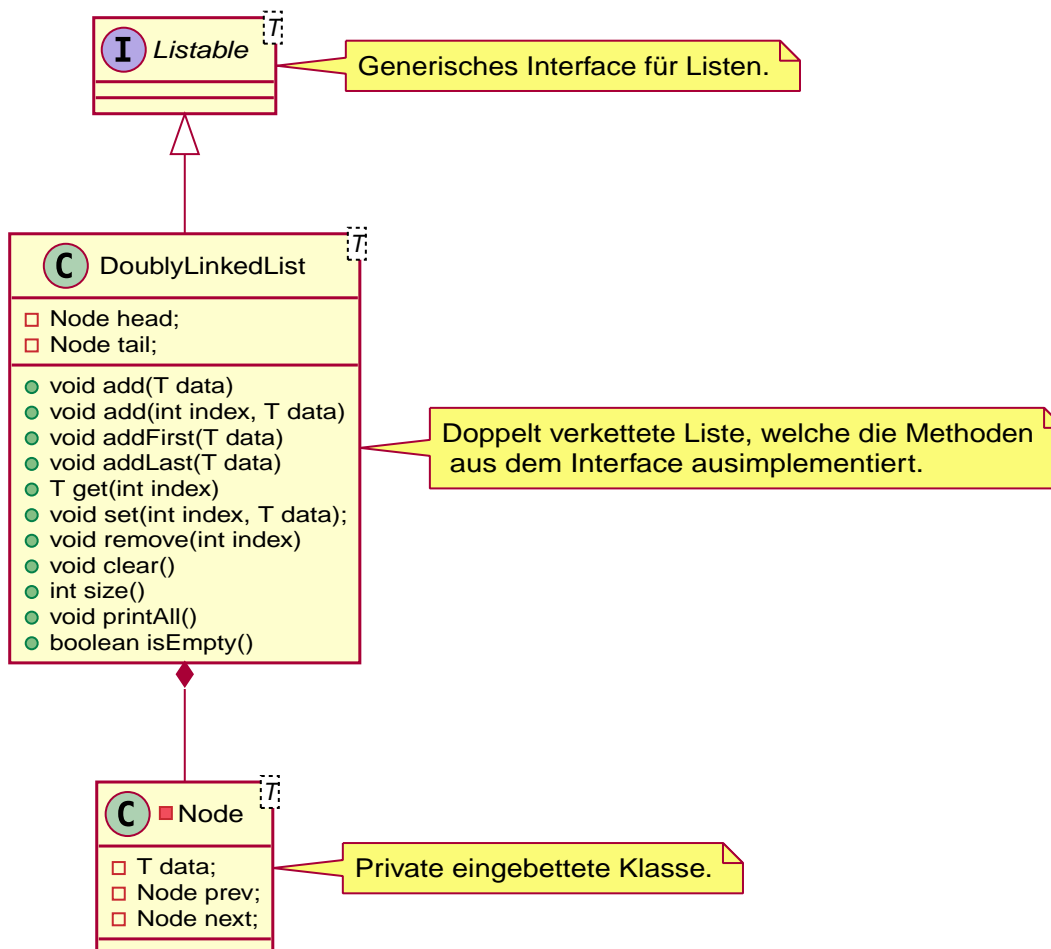
Nachdem Sie die Generierung der Methoden abgeschlossen haben, erweitern wir die Klasse **SinglyLinkedList<T>** mit der eingebetteten Klasse **Node** und fügen den Listenkopf **Node head** ein. Dieses Konstrukt wird für den Aufbau der **einfach verketteten Liste** benötigt.

```

1 public class SinglyLinkedList<T> implements Listable<T> {
2
3     Node head = null;        // Head of list
4
5     private class Node {      // Nested class
6         T data;
7         Node next;
8     }
9
10    @Override
11    public void add(T data) {
12        // TODO Auto-generated method stub
13        // Diese Methode wurde durch die IDE automatisch generiert mit einem
        Klick.
14    }
15
16    [...]
17 }

```

### 3. Step – Generische Klasse `DoublyLinkedList<T>` anlegen:



Als nächstes wird die Klasse `DoublyLinkedList` als **generische Klasse** angelegt.

```

1 package lists;
2
3 public class DoublyLinkedList<T> implements Listable<T> {
4
5     [...]
6 }

```

Diese implementiert das Interface `Listable<T>` mit dem **Schlüsselwort** `implements`.



Nachdem Sie das Interface mit `Listable<T>` zur Klasse korrekt hinzugefügt haben, können Sie die Glühbirne oder das Kontextmenü erneut verwenden, um alle **unimplemented Methods** mit nur einem Klick zu generieren. Diese Funktionalität gibt es in beiden IDEs Eclipse und IntelliJ.

Nachdem Sie die Generierung der Methoden abgeschlossen haben, erweitern wir die Klasse `DoublyLinkedList<T>` mit der eingebetteten Klasse `Node` und fügen den Listenkopf `Node head` und das Listenende `Node tail` ein. Dieses Konstrukt wird für den Aufbau der **doppelt verketteten Liste** benötigt.

```

1 public class DoublyLinkedList<T> implements Listable<T> {
2
3     Node head = null;    // Head of list
4     Node tail = null;    // Tail of list
5
6     private class Node {    // Nested class
7         T data;
8         Node next;
9         Node prev;
10    }
11
12    @Override
13    public void add(T data) {
14        // TODO Auto-generated method stub
15        // Diese Methode wurde durch die IDE automatisch generiert mit einem
        Klick.
16    }
17
18    [...]
19 }

```

#### 4. Step – Methoden der `SinglyLinkedList<T>` ausimplementieren:

- a. Als Erstes implementieren wir die Methode für das Hinzufügen eines Datenobjektes am Ende der einfach verketteten Liste.

```

1 @Override
2 public void addLast(T data) {
3     Node node = new Node();
4     node.data = data;
5     node.next = null;
6
7     if (head == null) {
8         head = node;
9     } else {
10        Node temp = head;
11        while (temp.next != null) {
12            temp = temp.next;
13        }
14        temp.next = node;
15    }
16 }

```



Niemals direkt auf Head operieren!

b. Als Nächstes implementieren wir die Methode für das Prüfen, ob die Liste leer ist.

```

1 @Override
2 public boolean isEmpty() {
3     return head == null;
4 }

```

c. Danach implementieren wir die Methode für das Ermitteln der Größe der einfach verketteten Liste.

```

1 @Override
2 public int size() {
3     int counter = 0;
4     Node temp = head;
5     while (temp != null) {
6         counter++;
7         temp = temp.next;
8     }
9     return counter;
10 }

```



Es gibt eine bessere Implementierung für das Ermitteln der Größe der Liste. An den richtigen Stellen muss eine Membervariable inkrementiert und dekrementiert werden, dann wird diese Methode effizienter. Die Effektivität liegt bereits bei beiden Implementierungen vor.

d. Darauf implementieren wir die Methode `get` für das Holen eines Datenobjektes.

```

1 @Override
2 public T get(int index) {
3     if ((head == null) || (index < 0) || (index >= size())) {
4         return null;
5     }
6     Node temp = head
7     for (int i = 0; i < index; i++) {
8         temp = temp.next;
9     }
10    return tmp.data;
11 }

```

- e. Als Nächstes implementieren wir die Methode für das Löschen aller Datenobjekte der einfach verketteten Liste.

```

1 @Override
2 public void clear() {
3     head = null;
4 }

```

- f. Im Anschluss implementieren wir die Methode für das Ausgeben der Datenobjekte der einfach verketteten Liste.

```


1 @Override
2 public void printAll() {
3     Node temp = head;
4     while (temp != null) {
5         System.out.println(temp.data);
6         temp = temp.next;
7     }
8 }

```



Die Methode `printAll` gehört nicht in das Interface `Listable<T>`. Das Thema Iteratoren wird an dieser Stelle nicht weiter behandelt. Iteratoren wären hier die saubere Lösung für das Iterieren über die Datenobjekte in der Liste. Dies wird in dieser Aufgabenstellung nicht verlangt.

5. **Step** – Datenobjekt `Student` anlegen:

 Student
<ul style="list-style-type: none"> <li>❑ String prename</li> <li>❑ String surname</li> <li>❑ int course</li> <li>❑ int matriculationNumber</li> </ul>
<ul style="list-style-type: none"> <li>● Student(String prename, String surname, int course, int matriculationNumber)</li> <li>● String toString()</li> <li>● String getSurname()</li> <li>● void setSurname(String Surname)</li> <li>● String getPrename()</li> <li>● void setPrename(String prename)</li> <li>● int getCourse()</li> <li>● void setCourse(int course)</li> <li>● int getMatriculationNumber()</li> <li>● void setMatriculationNumber(int matriculationNumber)</li> </ul>

Bevor wir die Liste testen können, müssen wir ein Datenobjekt anlegen. Wir legen uns hier das Datenobjekt **Student** an. Es können auch andere Datenobjekte von der Liste verwaltet werden, da diese generisch ist. Legen Sie bitte einen Studenten mit den folgenden Eigenschaften: Vorname, Nachname, Matrikelnummer und Kurs; an. Der Kurs soll später als **Enum** verwaltet werden, für unsere Testzwecke reicht aber **int** als **primitiver Datentyp** aus.

```

1 package data;
2
3 public class Student {
4     private String prename;
5     private String surname;
6     private int course;
7     private int matriculationNumber;
8
9     // Automatisch per IDE generiert - Generate Constructor using fields
10    // Automatisch per IDE generiert - Generate toString
11    // Automatisch per IDE generiert - Generate getter and setter
12
13    [...]
14 }

```



Der Konstruktor mit Parametern, die Getter/Setter und die toString-Methode für den Studenten können per IDE automatisch generiert werden, nachdem Sie die Eigenschaften (Membervariablen: Prenom, Surname, Course und Matriculation number) vom Studenten deklariert haben.

#### 6. Step – Testen der **SinglyLinkedList<T>** und **DoublyLinkedList<T>**:

Nachdem wir die Grundlagen für das Testen der Liste gelegt haben, können wir ein kleines Testprogramm schreiben, welches einen kleinen Teil der Funktionalitäten von unserer **einfach und doppelt verketteten Liste** testet.



```

1 import data.Student;
2 import lists.DoublyLinkedList;
3 import lists.Listable;
4 import lists.SinglyLinkedList;
5
6 public class MyApp {
7
8     public static void main(String[] args) {
9         run(new SinglyLinkedList<Student>());
10        run(new DoublyLinkedList<Student>());
11    }
12
13    private static void run(Listable<Student> students) {
14
15        students.addLast(readStudentsFromStdIn(students));
16
17        students.add(new Student("Sheldon", "Cooper", 67, 78));
18
19        Student s3 = new Student("Micky", "Maus", 67, 45);
20        students.addLast(s3);
21
22        printList(students);
23
24        students.clear();
25    }
26
27    private static Student readStudentsFromStdIn()
28    {
29        // TODO: Read values from System.in (Stdin)
30        Student student = new Student("Ted", "Mosby", 89, 89);
31        return student;
32    }
33
34    private static void printList(Listable<Student> students) {
35        System.out.println("Type: " + students.getClass().getSimpleName());
36        System.out.println("Size: " + students.size());
37        System.out.println("IsEmpty: " + students.isEmpty());
38        System.out.println("Students");
39        System.out.println("Get:" + students.get(0));
40        System.out.println("Get:" + students.get(1));
41        System.out.println("Get:" + students.get(2));
42        System.out.println("Get:" + students.get(-1));
43        System.out.println("Get:" + students.get(3));
44        students.printAll();
45    }
46 }

```

Vervollständigen Sie die Implementierung der einfach und doppelt verketteten Liste. Lesen Sie das Aufgabenblatt bzgl. der einfach und doppelt verketteten Liste genau durch.



Ab hier müssen Sie die Implementierung alleine vervollständigen und anpassen.

Folgend die komplette UML-Notation vom Testprogramm. Es handelt sich um ein Testprogramm, es muss für die Erfüllung der Aufgabe an mehreren Stellen angepasst werden. Hiermit ist der Grundstein für den Anfang des Aufgabenblattes gelegt. Sie sollten jetzt auch **Code-Snippets aus den vorherigen Aufgabenblätter wiederverwenden**, wie z. B. die Klasse **Console**.

