hhu.

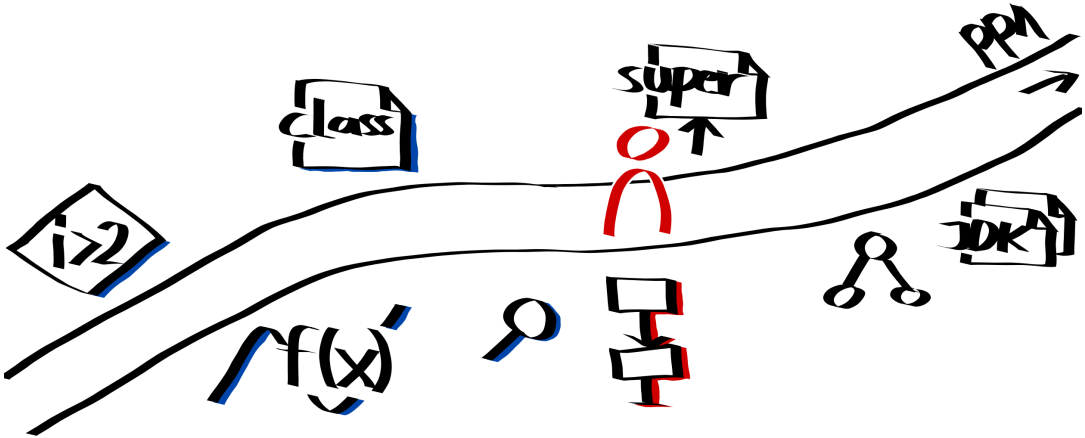
Kapitel 5: Listen & generische Datentypen

VL 18: Generics

Wintersemester 2023/24

Stand 7. Dezember 2023, 07:59 Uhr

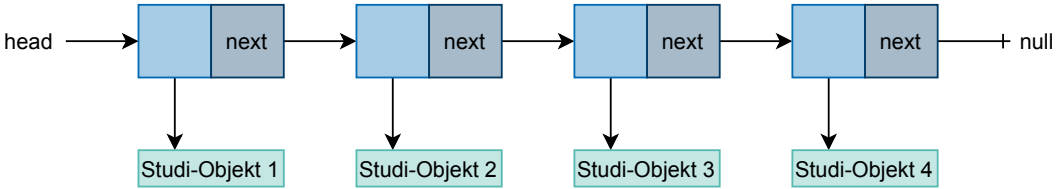
Wo stehen wir gerade?



Vorlesungsfeedback

Könnte man ein Liste für Objekte programmieren?

Ja!



Anpassungen am Code:

- Node-Klasse speichert Studi-Objekte statt int-Werte
- `insert` nimmt Studi-Objekt statt Integer
- ...

So könnten wir auch eine Liste für Bälle erstellen

Heinrich Heine Universität Düsseldorf

```java
public class BallList {
    private class Node {
        private Ball element = null;
        private Node next = null;

        private Node(Ball e, Node n) {
            this.element = e;
            this.next = n;
        }
    }

    private Node head = null;

    public void insert(Ball ball) {
        head = new Node(ball, head);
    }
}
```

```java
public class StudiList {
  private class Node {
    private Studi element = null;
    private Node next = null;

    private Node(Studi e, Node n) {
      this.element = e;
      this.next = n;
    }
  }

  private Node head = null;

  public void insert(Studi studi) {
    head = new Node(studi, head);
  }
}
```

Gibt es ein Problem bei diesem Ansatz?

Redundanz!

- Bräuchten für jede Klasse eine eigene Listen-Implementierung
- Lösung? Interfaces funktionieren nicht . . .
  - Typ-Information geht verloren
    - ⇒ können mit Listenelementen nur das machen, was das Interface vorgibt
    - ⇒ oder müssen zur Laufzeit casten (fehleranfällig)
  - bestehende Klassen (z. B. String) implementieren Interface nicht

! Mischen verschiedener Typen in derselben Liste möglich

Lösung: Generics[1]

- Idee: Variablen für Datentypen
- in Java: Generics
- Statt konkrete Variablen-Typen Verwendung von Typ-Variablen, z. B. `T`
- Deklaration der Variablen mit spitzen Klammern nach Klassenname

```
1  public class List<T> {
2
3      private class Node {
4          private T element = null;
5          private Node next = null;
6
7          private Node(T element, Node next) {
8              this.element = element;
9              this.next = next;
10         }
11     }
```

[1] Generische Typen (parametrischer Polymorphismus)

Verwendung von `T` in Methoden

Heinrich Heine Universität Düsseldorf

```java
public class List<T> {

    private Node head = null;

    public void insert(T object) {
        head = new Node(object, head);
    }

    public T first() {
        return head.element;
    }
}
```

Verwendung der generischen Liste

Heinrich Heine Universität Düsseldorf

```
1 List<String> stringList = new List<String>();
2 stringList.insert("World");
3 stringList.insert("Hello");
4
5 String firstElement = stringList.first();
6 System.out.println(firstElement); // "Hello"
```

- Angabe des konkreten Typs bei Deklaration und Konstruktoraufruf
  - konkreter Typ kann Klasse oder Interface sein
- Zeile 1 darf auch abgekürzt werden:
  `List<String> stringList = new List<>();`

Typvariablen und primitive Datentypen

Heinrich Heine Universität Düsseldorf

- nur Objekttypen für Typvariable einsetzbar
- folgendes funktioniert **nicht**:

```
List<int> intList = new List<int>();
```

- Lösung?

Typvariablen und primitive Datentypen

Heinrich Heine Universität Düsseldorf

- nur Objekttypen für Typvariable einsetzbar
- folgendes funktioniert **nicht**:
  `List<int> intList = new List<int>();`
- Lösung: Benutzung der Klasse Integer

```
11 List<Integer> intList = new List<Integer>();
12 intList.insert(1);
```

  - primitive Integer automatisch in Integer-Objekte umgewandelt (Autoboxing)
  - funktioniert analog für alle primitiven Datentypen
  - ! wie bei String-Objekten gilt: zwei Integer-Objekte mit gleichem Inhalte nicht unbedingt `==`

Benutzen die fertigen Listen im JDK auch Generics?

Heinrich Heine
Universität Düsseldorf

Benutzen die fertigen Listen im JDK auch Generics?

Heinrich Heine Universität Düsseldorf

Ja!

Sehen wir uns später noch genauer an . . .

```java
import java.util.List;
import java.util.LinkedList;

public class ListJdkDemo {

    public static void main(String[] args) {
        List<String> list = new LinkedList<String>();
        // LinkedList<T> implementiert das Interface List<T>

        list.add("Hello");
        String firstElement = list.get(0);
    }

}
```

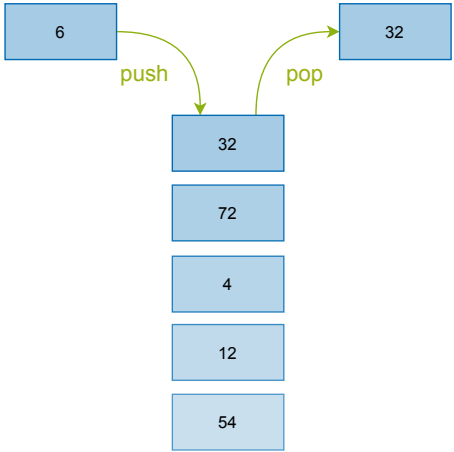
Stack

Heinrich Heine Universität Düsseldorf

Datenstruktur mit folgenden Operationen:

- `push`: fügt neues Element ein
- `pop`: entfernt zuletzt eingefügtes Element und gibt es zurück

→ Last-In-First-Out-Prinzip (LIFO)



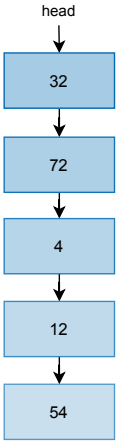
Stack

Heinrich Heine Universität Düsseldorf

Datenstruktur mit folgenden Operationen:

- `push`: fügt neues Element ein
- `pop`: entfernt zuletzt eingefügtes Element und gibt es zurück

→ Last-In-First-Out-Prinzip (LIFO)



Anwendungbeispiel: Expression-Parsing

Heinrich Heine
Universität Düsseldorf

- einfache Rechnungen von Standardeingabe lesen und ausführen (Interpreter):
  - `1 + 2` $\Rightarrow$ 3
  - `4 * 2` $\Rightarrow$ 8
- Vorschläge?

Anwendungbeispiel: Expression-Parsing

Heinrich Heine Universität Düsseldorf

- einfache Rechnungen von Standardeingabe lesen und ausführen (Interpreter):
  - `1 + 2` $\Rightarrow$ 3
  - `4 * 2` $\Rightarrow$ 8
- Vorschläge?

```java
Scanner stdin = new Scanner(System.in);

int a = stdin.nextInt();
String operator = stdin.next();
int b = stdin.nextInt();

switch(operator) {
    case "+":
        System.out.println(a + b);
        break;
    case "*":
        System.out.println(a * b);
        break;
}
```

Nächstes Level

- Ziel: Ausdrücke der Form `( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )`

[2] vereinfachte Form ohne Operatorpräzedenzen und mit expliziter Klammerung

Nächstes Level

- Ziel: Ausdrücke der Form `( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )`
- Shunting-yard-Algorithmus[2] (Edsger Dijkstra):
  - Operator?
    - → auf Operator-Stack pushen
  - Operand?
    - → auf Operanden-Stack pushen
  - öffnende Klammer
    - → ignorieren
  - schließende Klammer
    - → Operator *o* und zwei Operanden *a*, *b* popen, Ergebnis *a o b* auf Operanden-Stack pushen

[2] vereinfachte Form ohne Operatorpräzedenzen und mit expliziter Klammerung

Wie klappt das?

Heinrich Heine
Universität Düsseldorf

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden:
- Operatoren:

Wie klappt das?

Heinrich Heine
Universität Düsseldorf

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1
- Operatoren:

Wie klappt das?

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1
- Operatoren: +

Wie klappt das?

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1
- Operatoren: +

Wie klappt das?

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1
- Operatoren: +

Wie klappt das?

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1 2
- Operatoren: +

Wie klappt das?

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1 2
- Operatoren: + +

Wie klappt das?

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1 2 3
- Operatoren: + +

Wie klappt das?

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1 5
- Operatoren: +

(Aha, $(2 + 3)$ wird also quasi durch 5 ersetzt.)

Wie klappt das?

Heinrich Heine
Universität Düsseldorf

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1 5
- Operatoren: + *

Wie klappt das?

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1 5
- Operatoren: + *

Wie klappt das?

Heinrich Heine
Universität Düsseldorf

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1 5 4
- Operatoren: + *

Wie klappt das?

Heinrich Heine
Universität Düsseldorf

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1 5 4
- Operatoren: + * *

Wie klappt das?

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1 5 4 5
- Operatoren: + * *

Wie klappt das?

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1 5 20
- Operatoren: + *

(Aha, (4 · 5) wird also quasi durch 20 ersetzt.)

Wie klappt das?

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 1 100
- Operatoren: +

Wie klappt das?

Heinrich Heine
Universität Düsseldorf

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

- Operanden: 101
- Operatoren:

Bemerkungen

- Algorithmus klappt auch, wenn Operator nach Operanden (Reverse Polish Notation[3]):

```
( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
```

- Klammern sind dann redundant:

```
1 2 3 + 4 5 * * +
```

[3] Postfixnotation, im Gegensatz zur Präfixnotation des polnischen Mathematikers Jan Łukasiewicz

Bemerkungen

- Algorithmus klappt auch, wenn Operator nach Operanden (Reverse Polish Notation[3]):

```
( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
```

- Klammern sind dann redundant:

```
1 2 3 + 4 5 * * +
```

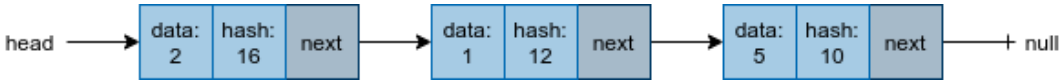
- Anwendungen:
  - Programmiersprachen: Postscript, JVM-Bytecode, ...
  - spezielle Taschenrechner

```
iconst_1
iconst_2
iconst_3
iadd
iconst_4
iconst_5
imul
imul
iadd
```

[3] Postfixnotation, im Gegensatz zur Präfixnotation des polnischen Mathematikers Jan Łukasiewicz

Anwendungbeispiel: Blockchain

- nur Einfügen auf einer Seite, kein Löschen
- zusätzlicher *Hashwert*$^{\rightarrow \text{später}}$, der sich aus Node-Inhalt und Vorgänger-Hash berechnet
  - ermöglicht Prüfung, ob doch (irgendwie) nachträglich Inhalte verändert wurden



(unsichere) Hashfunktion hier im Beispiel: $2 \cdot \text{data} + \text{Vorgänger-Hash}$

Lernziele

Heinrich Heine
Universität Düsseldorf

Sie können am Ende der Woche ...

- Generische Datentypen **erstellen**.
- Generische Datentypen **verwenden**.

Wichtige Begriffe

Generics <T> LinkedList<String>

new LinkedList<>();

Übungsblatt

- Listen und Arrays
- Verkettete Liste