Programmieren II

Dr. Klaus Höppner

Hochschule Darmstadt – Sommersemester 2010

Dynamische Listen: Stack

Generics

Motivation

Hauptteil dieser Vorlesung sind die so genannten *Generics*.

Zur Motivation (und als Vorbereitung der Datencontainer-Klassen für kommenden Vorlesungen) wird als Beispiel zunächst eine Klasse entwickelt, die einen Stapel (Stack) realisiert.

Einfache Liste: Ein Stapel (Stack)

Die einfachste Form einer Liste ist ein

Stapel (*stack*).

Analog zu einem Stapel Spielkarten sind mit einem Stack nur zwei Operationen möglich:

- eine neues Element oben auf den Stapel legen,
- das oberste Element vom Stapel herunter nehmen.

Bei einem Stack erhält man beim Herunternehmen eines Elementes jeweils das zuletzt hinzugefügte Element.
Dies nennt man *last in, first out* (LIFO).

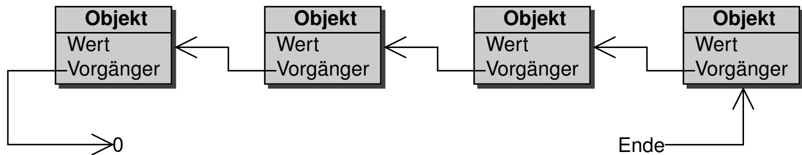
Hierin unterscheided sich der Stack von einer Queue (Warteschlange), bei der das Prinzip *first in, first out* (FIFO) gilt.

Ansatz

Für einen Stapel kann man folgenden Ansatz wählen:

- Man muss wissen, welches das oberste Element ist.
- Für jedes Element muss man wissen, welches das zuvor hinzugefügte (also das darunter liegende) ist.
- Fügt man ein neues Element hinzu, wird dieses zum neuen obersten Element, unter dem das bisherige oberste Element liegt.
- Nimmt man das oberste Element herunter, so wird das darunter liegende Element zum neuen obersten Element.
- Das unterste Element erkennt man daran, dass keins darunter liegt.

Schaubild



Klasse StackElement

```
public class StackElement {
    private int value;
    private StackElement next;

    public StackElement(int value, StackElement next) {
        this.value = value;
        this.next = next;
    }

    public int getValue() {
        return value;
    }

    public StackElement getNext() {
        return next;
    }
}
```

Klasse Stack

```
public class Stack {
    private StackElement top = null;

    public void push(int value) {
        StackElement newTop = new StackElement(value, top);
        top = newTop;
    }

    public int pop() {
        if (top==null)
            throw new RuntimeException("Leerer Stack");

        int value = top.getValue();
        top = top.getNext();
        return value;
    }
}
```

Analyse

Beim Anlegen einer Instanz von `Stack` ist das Attribut `top` der Null-Pointer. Der Stapel ist zu Anfang somit leer.

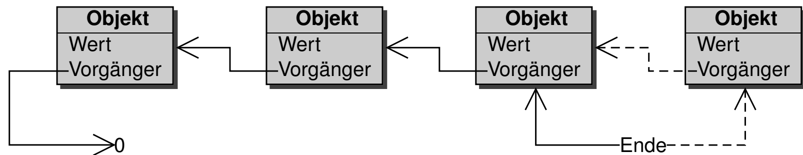
Beim Aufruf der Methode `push` wird eine neue Instanz der Klasse `StackElement` angelegt:

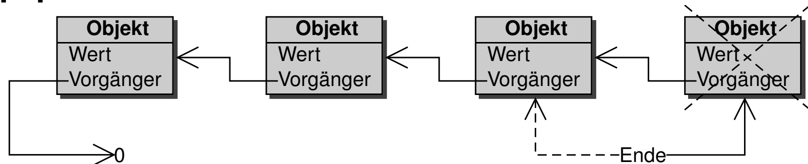
- das Attribut `value` ist der neu auf den Stapel gelegte Wert,
- das Attribut `next` wird auf das bisherige oberste Element gesetzt.

Anschließend verweist das Attribut `top` auf das neu angelegte Stapelelement.

Beim Aufruf der Methode `pop` wird sich der Wert des obersten Elementes gemerkt, dann das Attribut `top` auf den Vorgänger umgesetzt und der gemerkte Wert zurück gegeben.

Schaubild





Anwendung in main

```
public class App {
    public static void main(String[] args) {
        Stack stack = new Stack();

        stack.push(3);
        stack.push(5);
        stack.push(7);

        try {
            System.out.println(stack.pop());
            System.out.println(stack.pop());
            System.out.println(stack.pop());
            System.out.println(stack.pop());
        } catch (RuntimeException e) {
            System.out.println(e.getMessage());
        }
    }

}
```

Generics – Motivation

Im vorigen Beispiel wurde eine Klasse `Stack` vorgestellt, die einen Stapel für den Datentyp **`int`** repräsentiert.

Hierbei ist die Tatsache, dass es sich um den Typ **`int`** handelt, für die grundsätzliche Funktionalität der Klassen irreleveant. Daher würden die Klassen `Stack` und `StackElement` für Stapel für die Typen **`double`**, `String`, . . . praktisch identisch aussehen.

Möchte man nicht deckungsgleiche Klassen für viele verschiedene Typen definieren, bestünde die einfache Lösung darin, die o. a. Klassen allgemein für `Object` zu definieren.

Klasse StackElement (mit Object)

```
public class StackElement {
    private Object value;
    private StackElement next;

    public StackElement(Object value, StackElement next) {
        this.value = value;
        this.next = next;
    }

    public Object getValue() {
        return value;
    }

    public StackElement getNext() {
        return next;
    }
}
```

Klasse Stack (mit Object)

```
public class Stack {
    private StackElement top = null;

    public void push(Object value) {
        StackElement newTop = new StackElement(value, top);
        top = newTop;
    }

    public Object pop() {
        if (top==null)
            throw new RuntimeException("Leerer Stack");

        Object value = top.getValue();
        top = top.getNext();
        return value;
    }
}
```

Anwendung in main (mit Object)

```
public class App {
    public static void main(String[] args) {
        Stack stack = new Stack();

        stack.push("Hallo");
        stack.push("Welt");

        String s;
        for (;;) {
            try {
                s = (String) stack.pop();
                System.out.println(s);
            } catch (RuntimeException e) {
                System.out.println(e.getMessage());
                break;
            }
        }
    }
}
```

Probleme mit dem `Object`-Ansatz

Der zuvor beschriebene Ansatz liefert eine generelle Lösung für Stapel allgemeiner Typen. Tatsächlich ist dies die Art und Weise wie Datencontainer (z. B. `ArrayList`, `LinkedList`) bis Java 1.4 realisiert wurden.

Dieser Lösungsansatz hat aber entscheidende Nachteile:

- Aus einem Stapel, dem nur Strings hinzugefügt werden, werden mit `pop()` jeweilse nur Referenzen vom Typ `Object` zurück geholt. Es ist also ein expliziter Cast nötig, um hieraus wieder einen `String` zu machen:
  ```
  String s = (String) stack.pop();
  ```
- Es geht die Typsicherheit verloren. In einen Stapel können nun mit `push(...)` beliebige Objekte unterschiedlichen Typs geschoben werden, z. B.
  ```
  stack.push("Hallo"); stack.push(new Double(1.5));
  ```

Generics – Grundsätzliches

Generics sind eine Möglichkeit, in Java *parametrisierte Klassen* zu definieren, die einen generischen (nicht näher bestimmten) Datentyp verwenden, der erst beim Anlegen einer Instanz angegeben wird.
Das Grundprinzip lautet also:

- Bei der Definition der generischen Klasse bleibt der parametrisierte Datentyp offen (wobei dieser in der Definition auf eine bestimmte Menge beschränkt werden kann, so genannte *constraints*)
- Beim Anlegen einer Instanz mit **`new`** wird der parametrisierten Datentyp durch einen konkreten ersetzt (der eventuelle Constrains erfühlen muss)

Generics stehen in Java seit der Version 5.0 (Java 1.5) zur Verfügung.

Klasse StackElement (mit Generics)

```
public class StackElement<T> {
    private T value;
    private StackElement<T> next;

    public StackElement(T value, StackElement<T> next) {
        this.value = value;
        this.next = next;
    }

    public T getValue() {
        return value;
    }

    public StackElement<T> getNext() {
        return next;
    }
}
```

Klasse Stack (mit Generics)

```
public class Stack<T> {
    private StackElement<T> top = null;

    public void push(T value) {
        StackElement<T> newTop = new StackElement<T>(value, top);
        top = newTop;
    }

    public T pop() {
        if (top==null) throw new RuntimeException("Leerer Stack");
        T value = top.getValue();
        top = top.getNext();
        return value;
    }
}
```

Anwendung in main (mit Generics)

```
public class App {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<String>();

        stack.push("Hallo");
        stack.push("Echo");

        for (;;) {
            try {
                String s = stack.pop();
                System.out.println(s +
                        "; Länge: " + String.valueOf(s.length()));
            } catch (RuntimeException e) {
                break;
            }
        }
    }

}
```

Interne Realisierung

Man unterscheidet in OO Programmiersprachen zwei Arten der Realisierung von generischen Klassen:

Heterogene Variante Hier wird für *jeden* verwendeten Typ eine eigene Variante der Klasse kompiliert.

Homogene Variante Hier wird nur *eine* Variante der Klasse (i. A.) für den Typ `Object` kompiliert, und für die konkreten Typen werden dann automatisch entsprechende Casts beim Verwenden der Methoden eingebaut.

Java verwendet hierbei die *homogene* Variante, z. B. C++ hingegen die *heterogene*.

Konsequenzen

Die homogene Variante hat den Vorteil, dass nur einmal ein Byte-Code für die Klasse erzeugt werden muss (und dann für andere Anwendungen z. B. als Bibliothek in Form einer JAR-Datei zur Verfügung stehen kann).
In C++ sind hingegen keine Template-Bibliotheken möglich, da Maschinencode erst erzeugt wird, wenn der konkrete Typ bekannt ist, und dafür der Quellcode sichtbar sein muss.

Da der Java-Compiler alle Typinformationen entfernt (*type erasure*), ist zur Laufzeit nur noch die allgemeine Klasse bekannt. Parametrisierte Klassen zu unterschiedlichen Klassen können daher nicht unterschieden werden.

Beispiel

```
Stack<Double> dstack = new Stack<Double>();
Stack<String> sstack = new Stack<String>();

if (dstack instanceof Stack<Double>) { // Fehler!
    // ...
}

if (sstack instanceof Stack) { // korrekt
    // ...
}

if (dstack.getClass() == sstack.getClass()) {
    // wahr!
}
```

Gefahr: Verwenden von *raw types*

Da zur Laufzeit nur die *rohe* Klasse bekannt ist, kann man den konkreten Typ „durch die Hintertür" umgehen (*solange man die entsprechende Warnung ignoriert*):

```
Stack<Double> dstack = new Stack<Double>();
dstack.push(new Double(3.1));

Stack rawStack = dstack;
rawStack.push("Hallo");

System.out.println(dstack.pop().doubleValue());
System.out.println(dstack.pop().doubleValue());
```

Hier kommt es zu einer *ClassCastException*, da versucht wird, den in `dstack` „geschmuggelten" String als Double zu verwenden.

Constraints für parametrisierte Typen

Der Java-Compiler ersetzt beim Erzeugen des Byte-Codes den generischen Datentyp durch `Object`.

Dies bedeutet, dass in den Methoden der Klasse für entsprechende Werte nur die Methoden von `Object` benutzt werden können.
Dies reicht im bisherigen Beispiel einer Klasse für Stapel aus, da hier die Objekte des parametrisierten Typs nur gespeichert werden und mit diesen keine Aktionen durchgeführt werden.

Constraints für parametrisierte Typen (Forts.)

Beispiel: Soll in einer generischen Klasse eine Reihe von Integer-, Double-, ...-Werten addiert und hieraus der Mittelwert berechnet werden, so reicht die Klasse `Object` als Grundtyp nicht aus, da die Elemente als Zahlen addiert werden müssen.

In diesem Fall kann als Typ-Einschränkung (*constraint*) angegeben werden, dass nur Typen erlaubt sind, die von `Number` erben. Dann können alle Methoden aus `Number` verwendet werden, z. B. `doubleValue`.

Realisierung: Klasse MittelWert

```
public class MittelWert<T extends Number> {
    private double summe = 0.0;
    private int count = 0;

    public void add(T value) {
        summe += value.doubleValue();
        count++;
    }

    public double getAverage() {
        if (count==0)
            throw new RuntimeException("Mittelwert nicht definiert
        return summe/count;
    }
}
```

Realisierung: Klasse Main

```
public class Main {
    public static void main(String[] args) {
        MittelWert<Integer> l = new MittelWert<Integer>();

        l.add(new Integer(3));
        l.add(new Integer(4));
        l.add(new Integer(6));

        System.out.println(l.getAverage());
    }
}
```

Wildcards

Manchmal möchte man Referenzen auf generische Klassen verwenden, bei denen einem der konkrete Typ „egal“ ist. Dies kann man durch Verwenden von Wildcards mit ? erreichen.

Beispiel:

```
Stack<Double> dstack = new Stack<Double>();
Stack<Integer> istack = new Stack<Integer>();

Stack<?> gstack;

gstack = dstack;
// Aktionen ...
gstack = istack;
// Aktionen ...
```

Wildcards können insbesondere bei Parameterdefinitionen von Methoden hilfreich sein.

Beispiel

```
public class MittelWert<T extends Number> {
    ...

    public static MittelWert<Number>
        fromStack(Stack<? extends Number> stack) {

        MittelWert<Number> m = new MittelWert<Number>();
        for (;;) {
            try {
                m.add(stack.pop());
            } catch (RuntimeException e) {
                break;
            }
        }
        return m;
    }
}
```

Beispiel (Forts.)

Durch die statische Methode `fromStack(...)` mit Wildcard kann nun der Mittelwert aller Elemente eines numerischen Stapels berechnet werden (als Nebeneffekt wird der Stapel dabei geleert).

```
public class App {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<Integer>();

        stack.push(new Integer(3));
        stack.push(new Integer(4));

        MittelWert<Number> m = MittelWert.fromStack(stack);
        System.out.println(m.getAverage());
    }

}
```