5. Zyklen und co

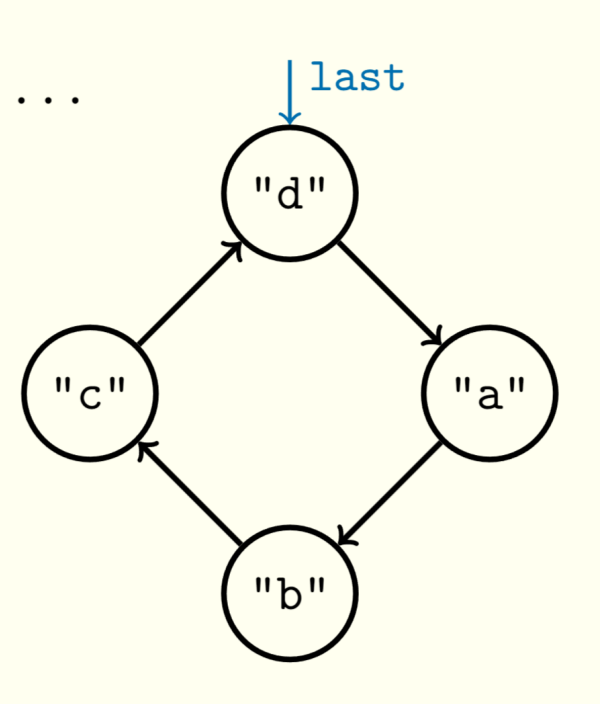
Quelle: ep2-05_Zyklen_doppelte-Verkettung_Abstraktionshierarchien_Objektschnittstellen.pdf
Beinhaltet: Zyklen, doppelte Verkettung Abstraktionshierarchien Objektschnittstellen

---

Ringlisten

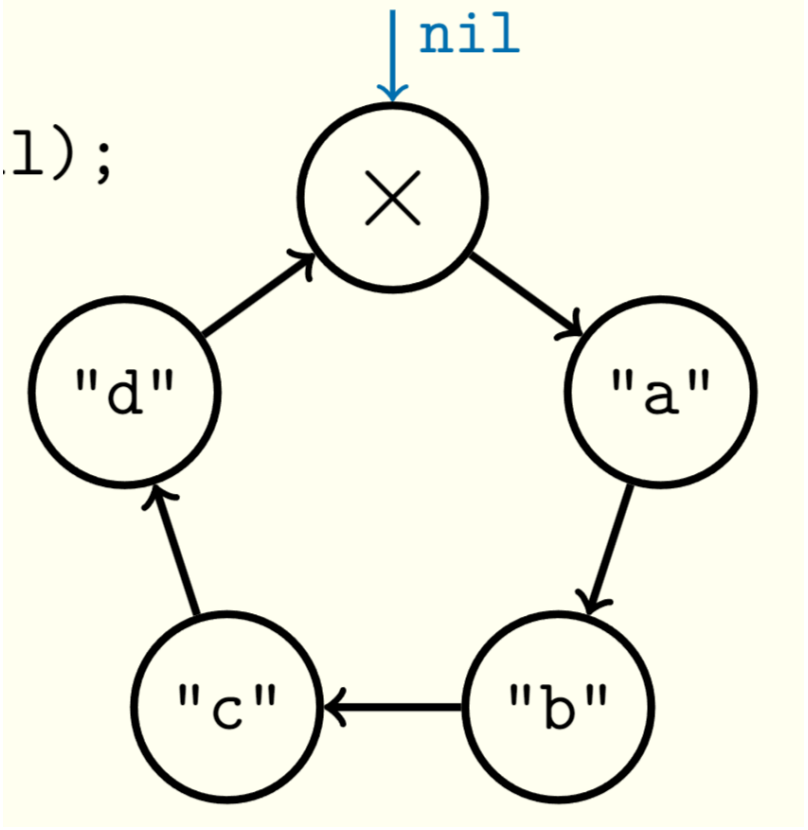
sind eine Erweiterung von 4. Lineare Liste und co > Lineare Liste mit der Extra Eigenschaft, dass das letzte Element entweder auf nil oder auf das erste Element referenziert.

```
public class RingQueue { private ListNode last; ...
    public String poll() {
        if (last != null) {
            ListNode n = last.next();
            if (n == last) {
                last = null;
            } else {
                last.setNext(n.next());
            } return n.value();
        } else {
        return null;
        }
    }

public void add(String v) {
    if (last == null) {
        last = new ListNode(v, null);
        last.setNext(last);
    } else {
    last.setNext(last = new ListNode(v, last.next()));
    }
}
```



Ringliste mit speziellem Knoten nil

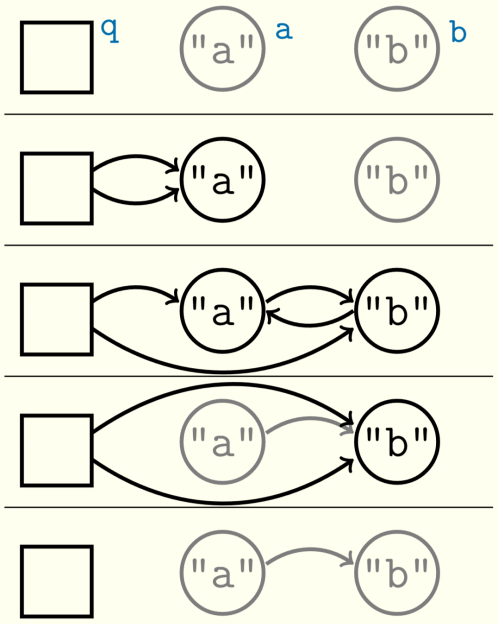
```
public class RingQueue {
    private ListNode nil = new ListNode(null, null);
    public RingQueue() {
        nil.setNext(nil);
    } ...

    public String poll() {
        ListNode n = nil.next();
        nil.setNext(n.next());
        return n.value();
    }
    public void add(String v) {
    nil.setValue(v);
     nil.setNext(nil = new ListNode(null, nil.next()));
    }
}
```


nil
"a"
"b"
"c"
"d"

Double linked lists

Sind eine weitere Erweiterung von 5. Zyklen und co > Ringlisten oder 4. Lineare Liste und co > Lineare Liste, mit dem unterschied, dass jedes Objekt seinen Nachfolger **aber auch den Vorgänger** referenziert.

ohne nil



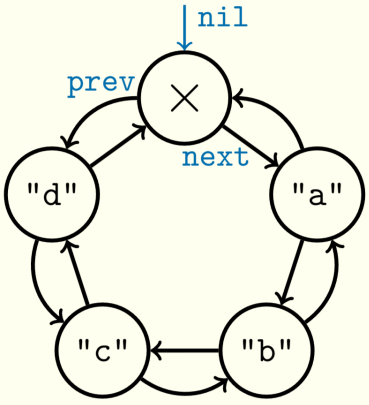
q ist Queue (abstrakter Datentyp), anfangs leer

```
q.add(a)    ≈ {q.head = a; q.last = a;}
```

eigener Zweig für Einfügen in leere Queue

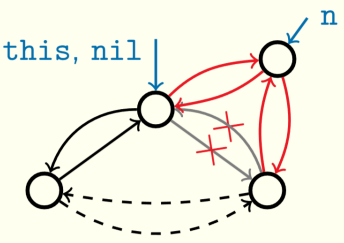
```
q.add(b)    ≈ {b.previous = q.last;
               q.last = q.last.next = b;}

q.poll()    ≈ {q.head = q.head.next;}

q.poll()    ≈ {q.head = q.last = null;}
```

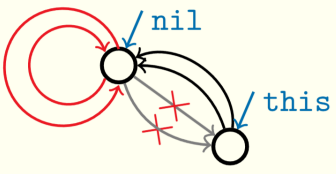
Mit nil



Traversierung in beide Richtungen
(z.B. Double-Ended-Queue)

spart viele Fallunterscheidungen

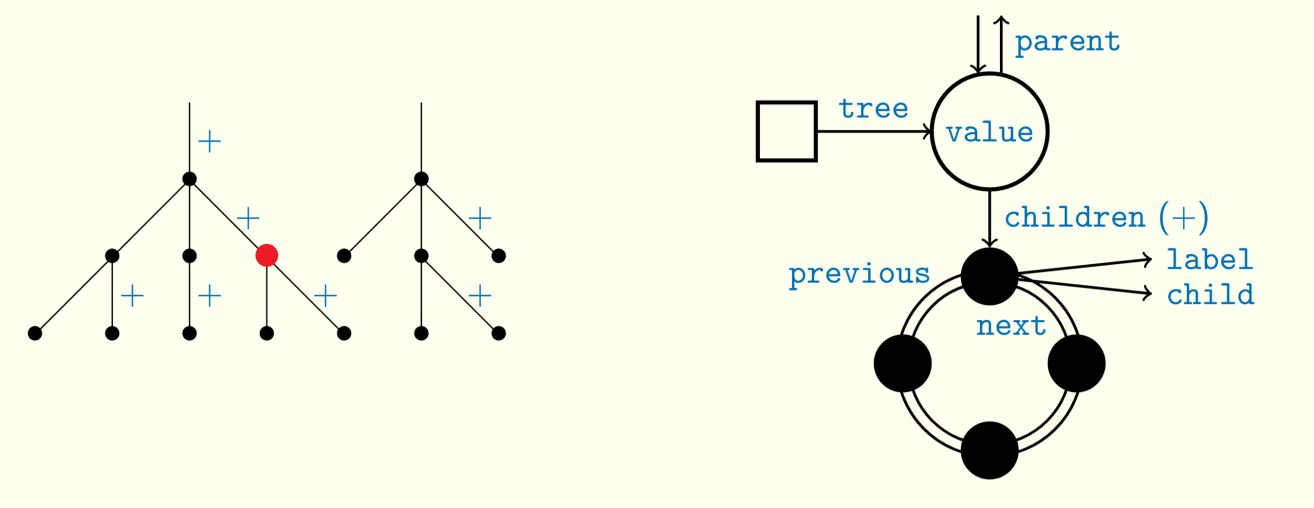




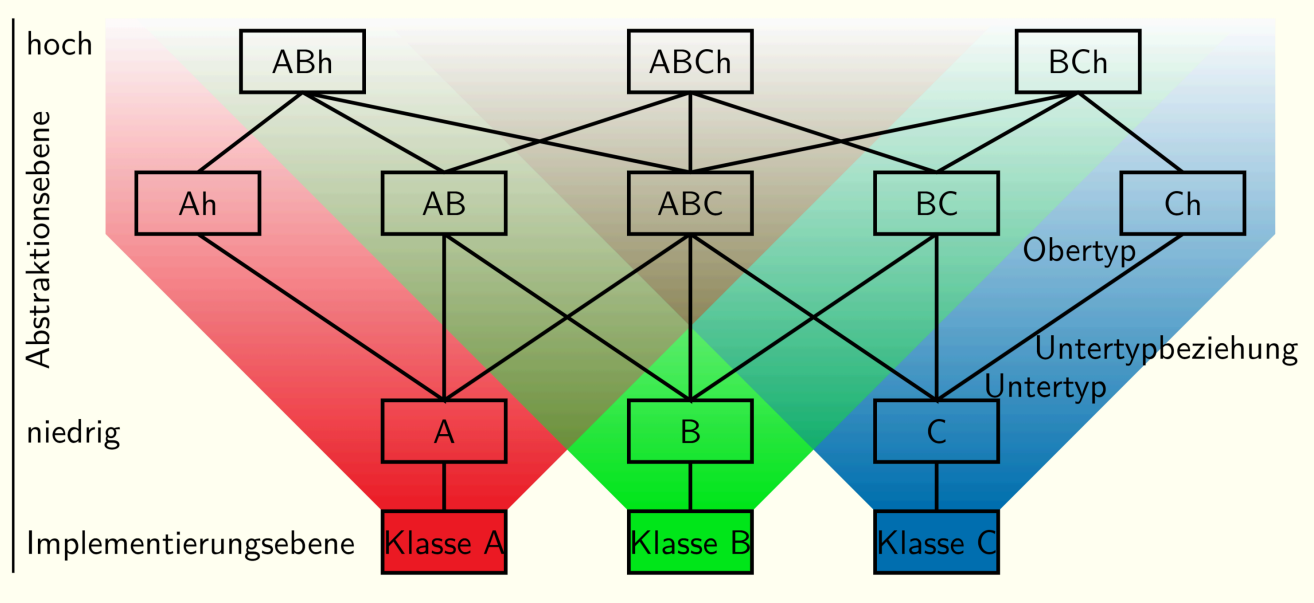

```
nil.add(...)
-->
DLNode n = new DLNode(
       ..., this, next);
next = next.prev = n;
```

```
nil.next.remove()
-->
next.prev = prev;
prev.next = next;
```

Navigieren durch verallgemeinerten Baum



Abstraktionshierarchie – Beziehungen zw. abstrakten Datentypen



Java-Interface zur Beschreibung einer Objektschnittstelle

```
/*******************************************************
BoxedText: Rectangular text within border lines.
public methods:
    void newDimensions(int width, int height);
    void setLine(int index, String txt);
    void print();
    String toString();
*******************************************************/
public interface AbstrBoxed {
    void newDimensions(int width, int height);
    void setLine(int index, String txt);
    void print(); String toString(); }
```

Da steht dann drinnen was die Klasse macht oder so

Java-Interface: Definition, Implementierung und Verwendung

```
public interface AbstrBoxed {
    void newDimensions(int width, int height);
    void setLine(int index, String txt);
    void print();
    String toString();
}
public class BoxedText implements AbstrBoxed {
... // defines at least all methods of AbstrBoxed
}
    AbstrBoxed ab = new BoxedText();
    ab.newDimensions(5, 3);
    ab.setLine(1, "ABCDE");
    ab.print();
```

Abstraktion auf höherer Ebene

```
public interface SetBoxed {
    void newDimensions(int width, int height);
```

```
    void setLine(int index, String txt);
}
public class BoxedText implements SetBoxed {
... // all methods including print and toString }
    BoxedText bt = new BoxedText(); bt.print(); // OK, BoxedText specifies
print
    SetBoxed sb = bt; // OK, BoxedText is subtype of SetBoxed
    sb.newDimensions(5, 3); // OK, SetBoxed specifies newDimensions
    sb.setLine(1, "ABCDE"); // OK, SetBoxed specifies setLine
    sb.print(); // ERROR, SetBoxed does not specify print bt = sb; // ERROR,
SetBoxed is no subtype of BoxedText
```

Klasse implementiert mehrere Interfaces

```
public interface Print {
    void print();
}
public class BoxedText implements SetBoxed, Print {
    ... // defines at least all methods of SetBoxed and Print
}

    BoxedText bt = new BoxedText();
    SetBoxed sb = bt;        // OK
    Print p = bt;            // OK
    sb.newDimensions(5, 3); // OK
    sb.setLine(1, "ABCDE"); // OK
    p.print();               // OK
    sb.print();              // ERROR
    p.setLine(1, "ABCDE");  // ERROR
```

Interface erweitert mehrere Interfaces

```
public interface SetBoxed {
    void newDimensions(int width, int height);
    void setLine(int index, String txt);
}
public interface Print {
    void print();
}
public interface AbstrBoxed extends SetBoxed, Print {
    void newDimensions(int width, int height);
    void print();
    String toString();
}
public class BoxedText implements AbstrBoxed {
    ...
}
```

Nominales Subtyping

Untertypbeziehungen beruhen auf Typdefinitionen (implements und extends), Vorhandensein von Methoden nicht hinreichend (`Print` $\neq$ `PrintBoxed`)

```
public interface Print {
    void print(); // prints 'this'
}
public interface PrintBoxed extends Print {
    void print(); // prints 'this' as text in a box
}

    Print p = ...;
    PrintBoxed pb = ...;
    p = pb;              // OK
    pb = p;              // ERROR
```

Vererbung auf Klassen

```
// every class inherits from exactly one other class,
// inherits from 'Object' if there is no 'extends',
// all public methods of superclass available in subclass
public class BoxedText extends Object implements AbstrBoxed {
    ... // exactly the same as without 'extends Object'
}

// public methods of BoxedText available in BoxedTextReset,
// BoxedTextReset is subtype of BoxedText
public class BoxedTextReset extends BoxedText {
    public void reset() {
        newDimensions(0, 0);
    }
}
```

Ersetzbarkeit

Objekt eines Untertyps überall verwendbar wo Objekt eines Obertyps erwartet

jede Referenzvariable (auch Parameter) hat gleichzeitig

- **deklarierten Typ:** Typ in der Deklaration der Variablen
- **dynamischen Typ:** Klasse des Objekts, das gerade in der Variablen steht

→ jeder Ausdruck hat:

- statisch vom Compiler ermittelten deklarierten Typ
- zur Laufzeit abfragbaren, sich mit Zuweisungen ändernden dynamischen Typ

mehr zu Ersetzbarkeit: 6. Ersetzbarkeit und co > Ersetzbarkeit

---

getClass, class, instanceof

```
Print p = new BoxedText();   // declared as Print, dynamic type BoxedText
Class cp = p.getClass();     // representation of dynamic type of p
Class cBT = BoxedText.class; // representation of class BoxedText
Class cP = Print.class;      // representation of interface Print
Class cA = Print[].class;    // representation of array of Print
Class ci = int.class;        // representation of int (no reference type)

cp == cBT                   -> true  (BoxedText is dynamic type of p)
cp == cP                    -> false (Print is not dynamic type of p)
p instanceof Print          -> true  (BoxedText is subtype of Print)
p instanceof BoxedText      -> true  (every type is subtype of itself)
p instanceof SetBoxed       -> true  (BoxedText is subtype of SetBoxed)
p instanceof BoxedTextReset -> false (not subtype of BoxedTextReset)
null instanceof Print       -> false (null is not subtype of any type)
```

Cast auf Referenztypen

```
Print p = new BoxedText();
p.print();                      // OK
p.setLine(0, "");               // syntax error, no setLine in Print
((BoxedText)p).setLine(0, "");  // OK, cast changes declared type
((SetBoxed)p).setLine(0, "");   // OK
((BoxedTextReset)p).print();    // error at runtime,  dynamic type
                                // BoxedText no subtype of BoxedTextReset
((Print)null).print();          // null.print -> error at runtime
p = (Print)null;                // OK, null can be cast to each ref. type
```