

5. Zyklen und co

Quelle: [ep2-05_Zyklen_doppelte-](#)

[Verkettung_Abstraktionshierarchien_Objektschnittstellen.pdf](#)

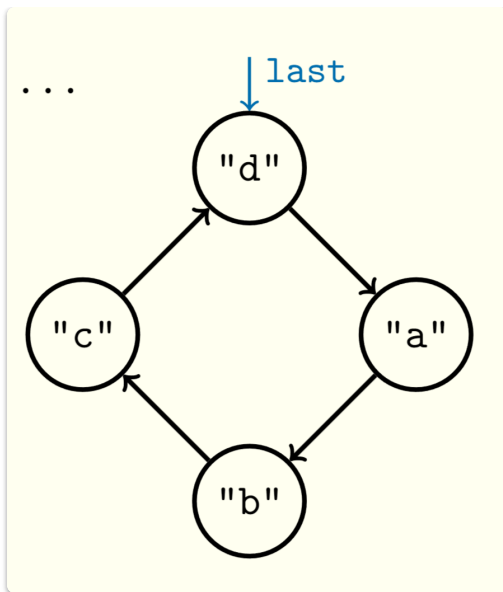
Beinhaltet: Zyklen, doppelte Verkettung Abstraktionshierarchien Objektschnittstellen

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()));
        }
    }
}
```



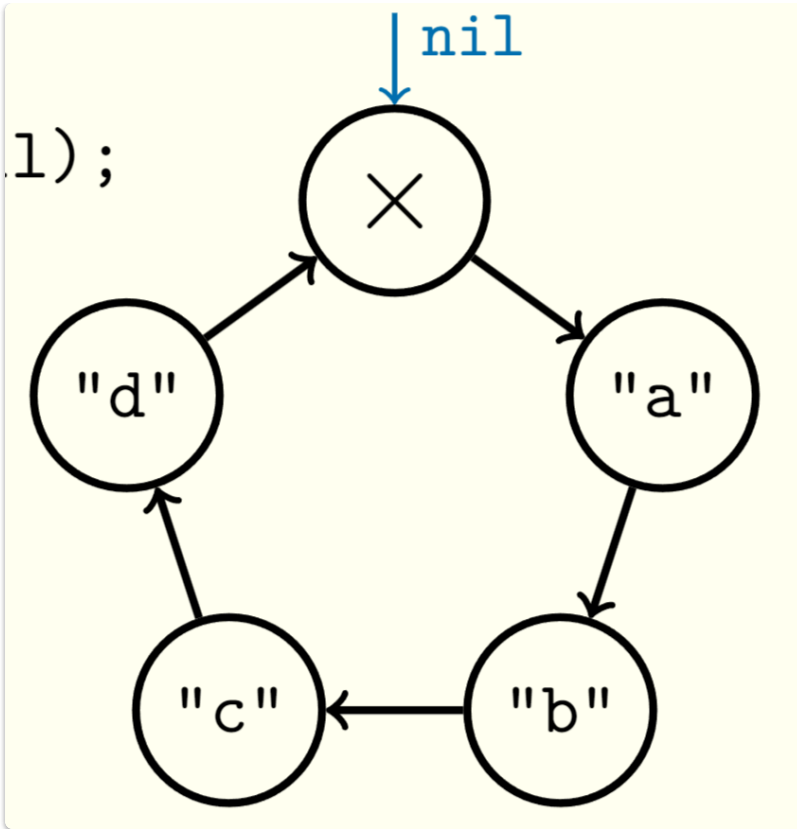
Ringliste mit speziellem Knoten nil

```

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()));
    }
}

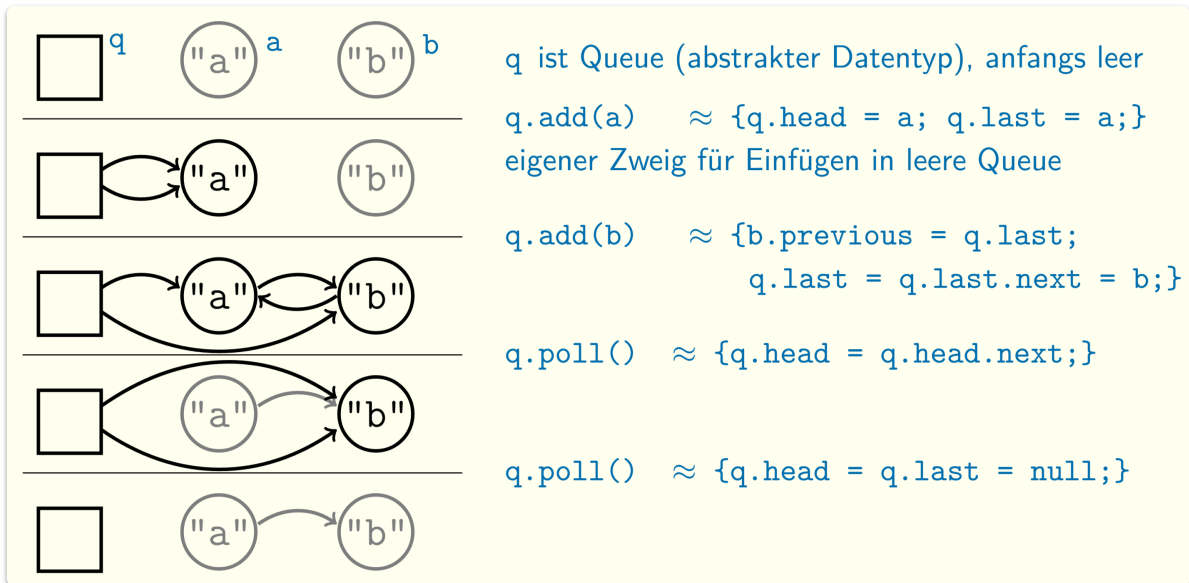
```



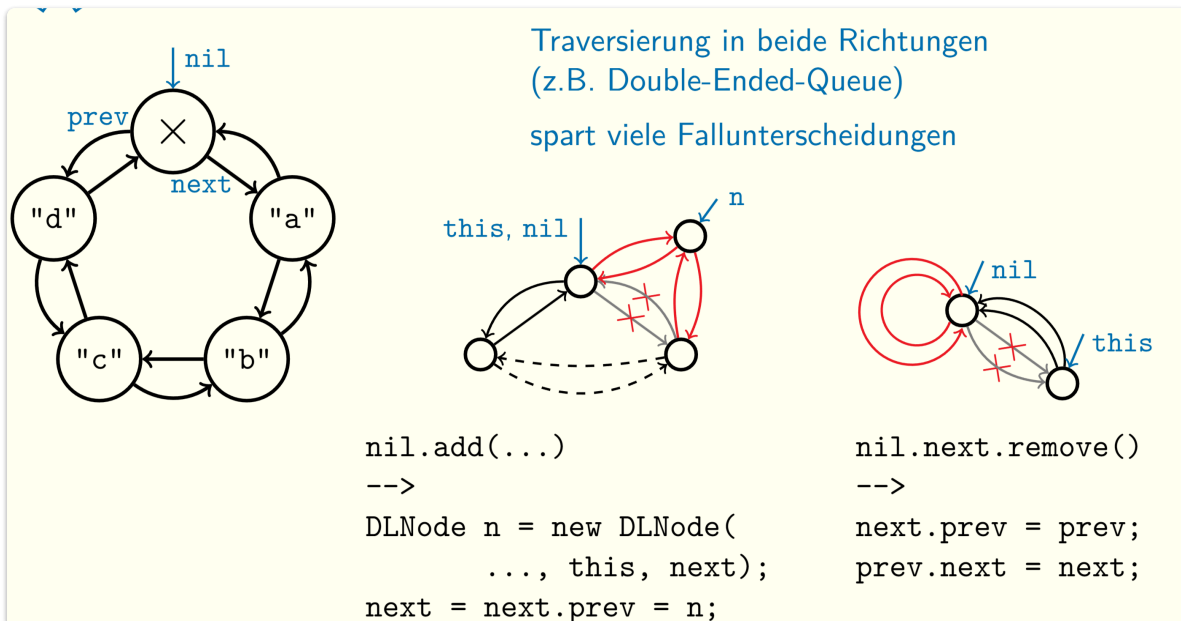
Double linked lists

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.

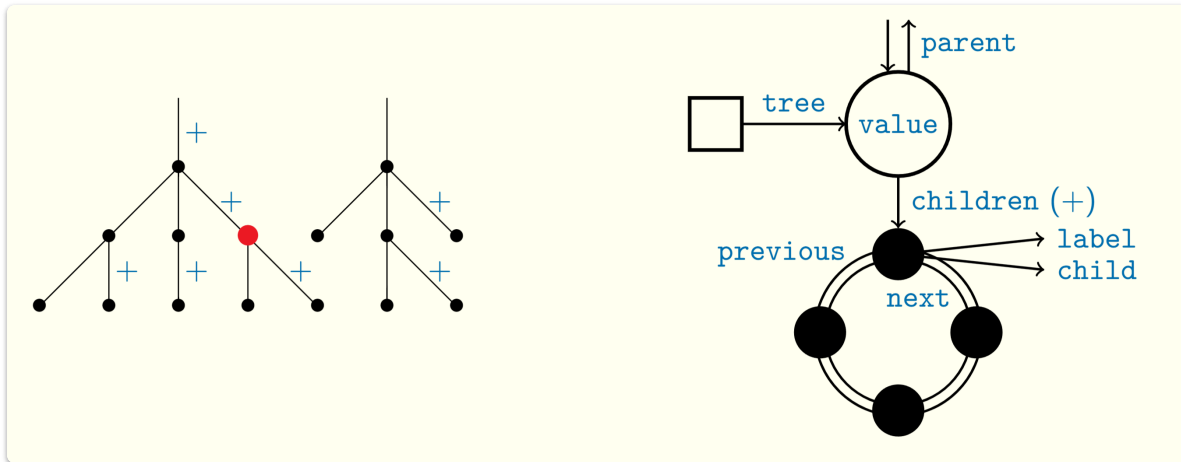
ohne nil



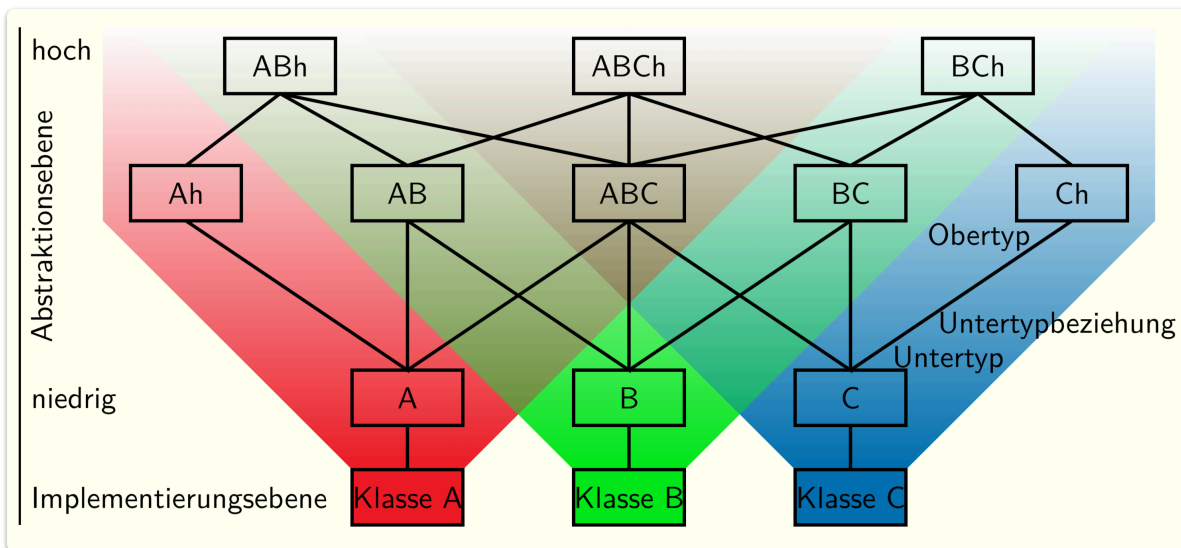
Mit nil



Navigieren durch verallgemeinerten Baum



Abstraktionshierarchie – Beziehungen zw. abstrakten Datentypen



A, B und C sind *unterschiedlich* und haben nichts miteinander zu tun. Aber es *kann sein, dass die Gemeinsamkeiten haben*. Und all das was miteinander zu tun hat liegt in einer neuen Abstraktionsebene. Natürlich müssen da auch die Beschreibungen die da zusammengehören zusammenpassen.

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();
}

```



Hier stehen die Methodenköpfe drinnen. Die Methoden sind dann automatisch Public. Also Interface ist eine Sichtfläche von einzelnen Methoden nach außen.

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
```



Da es das print nicht in sb gibt, kann man es hier nicht verwenden. Wenn man aber ein neues Interface hinzufügt, welches print public macht, ist das möglich.

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

Grundprinzip: Jede Klasse in Java erbt von *genau einer* anderen Klasse.

`extends` -Schlüsselwort:

- Wird verwendet, um die Oberklasse (Superklasse oder Elternklasse) einer Klasse festzulegen.
- **Syntax:** `public class Unterklasse extends Oberklasse { ... }`
- **Implizite Oberklasse:** Wenn keine `extends` -Klausel angegeben wird, erbt die Klasse automatisch von der Klasse `Object`.

Beispiel 1:

```
public class BoxedText extends Object implements AbstrBoxed {
    // ...
}
```

- `BoxedText` erbt von `Object`.
- Die `extends Object` -Klausel ist hier explizit, aber **redundant**, da `Object` die Standard-Oberklasse ist.
- `BoxedText` implementiert zusätzlich das Interface `AbstrBoxed`.

Verfügbarkeit von Methoden:

- Alle `public` -Methoden der Oberklasse sind automatisch in der Unterklasse verfügbar (werden vererbt).

Subtyping:

- Eine Unterklasse ist ein **Subtyp** ihrer Oberklasse. Das bedeutet, ein Objekt der Unterklasse kann überall dort verwendet werden, wo ein Objekt der Oberklasse erwartet wird.

Beispiel 2:

```
public class BoxedTextReset extends BoxedText {
    public void reset() {
        newDimensions(0, 0);
    }
}
```

- `BoxedTextReset` erbt von `BoxedText`.
- Alle `public` -Methoden von `BoxedText` (und somit auch die geerbten `public` -Methoden von `Object`) sind in `BoxedTextReset` verfügbar.

- `BoxedTextReset` fügt eine eigene `public`-Methode `reset()` hinzu.
- `BoxedTextReset` ist ein Subtyp von `BoxedText`. Das heißt, eine Variable vom Typ `BoxedText` könnte ein Objekt vom Typ `BoxedTextReset` referenzieren.

Zusammenfassend:

Vererbung ermöglicht die Schaffung neuer Klassen, die auf bestehenden Klassen aufbauen, wodurch Code-Wiederverwendung gefördert und eine hierarchische Struktur von Klassen geschaffen wird. Die Unterklasse erbt die öffentlichen Eigenschaften und Verhaltensweisen ihrer Oberklasse und kann diese erweitern oder spezialisieren.

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
```