

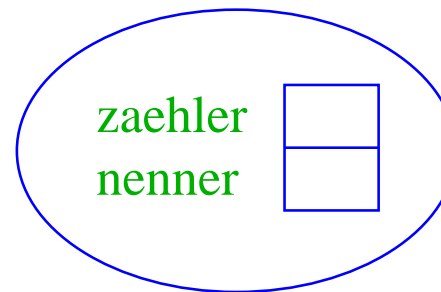
## 9 Klassen und Objekte

Datentyp	=	Spezifikation von Datenstrukturen
Klasse	=	Datentyp + Operationen
Objekt	=	konkrete Datenstruktur

## Beispiel: Rationale Zahlen

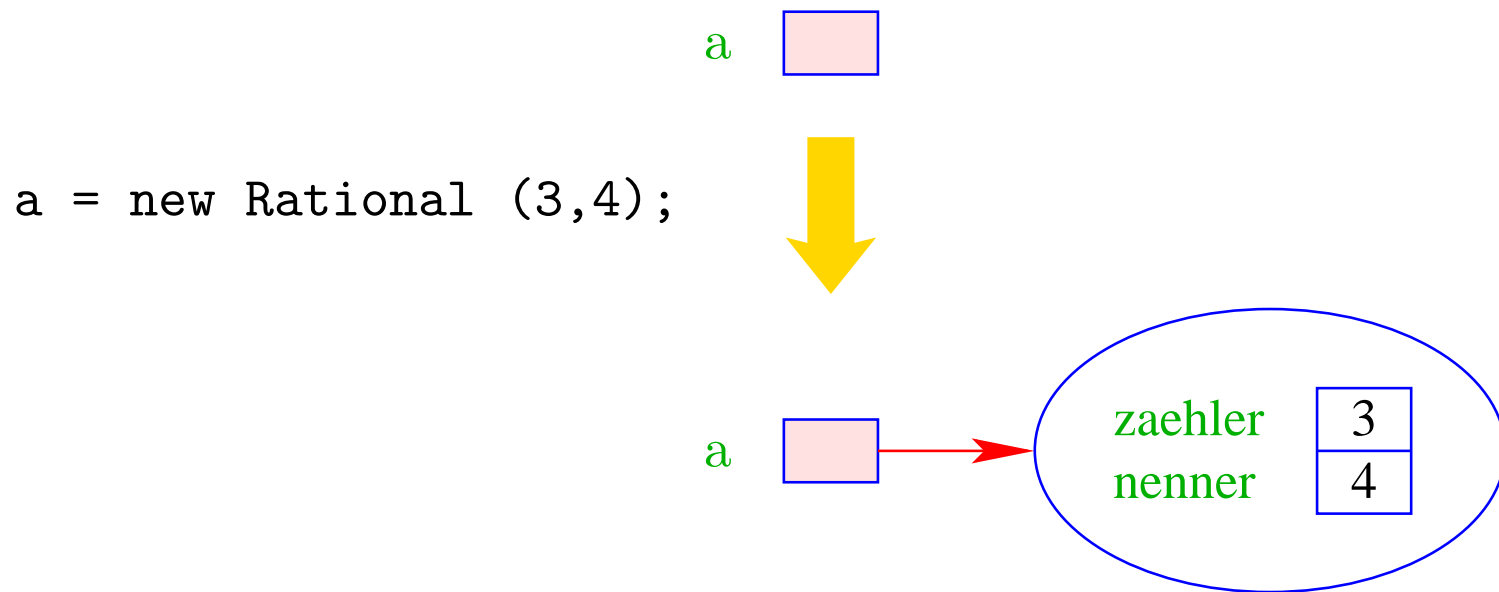
- Eine rationale Zahl  $q \in \mathbb{Q}$  hat die Form  $q = \frac{x}{y}$ , wobei  $x, y \in \mathbb{Z}$ .
- $x$  und  $y$  heißen Zähler und Nenner von  $q$ .
- Ein Objekt vom Typ `Rational` sollte deshalb als Komponenten `int`-Variablen `zaehler` und `nenner` enthalten:

Objekt:



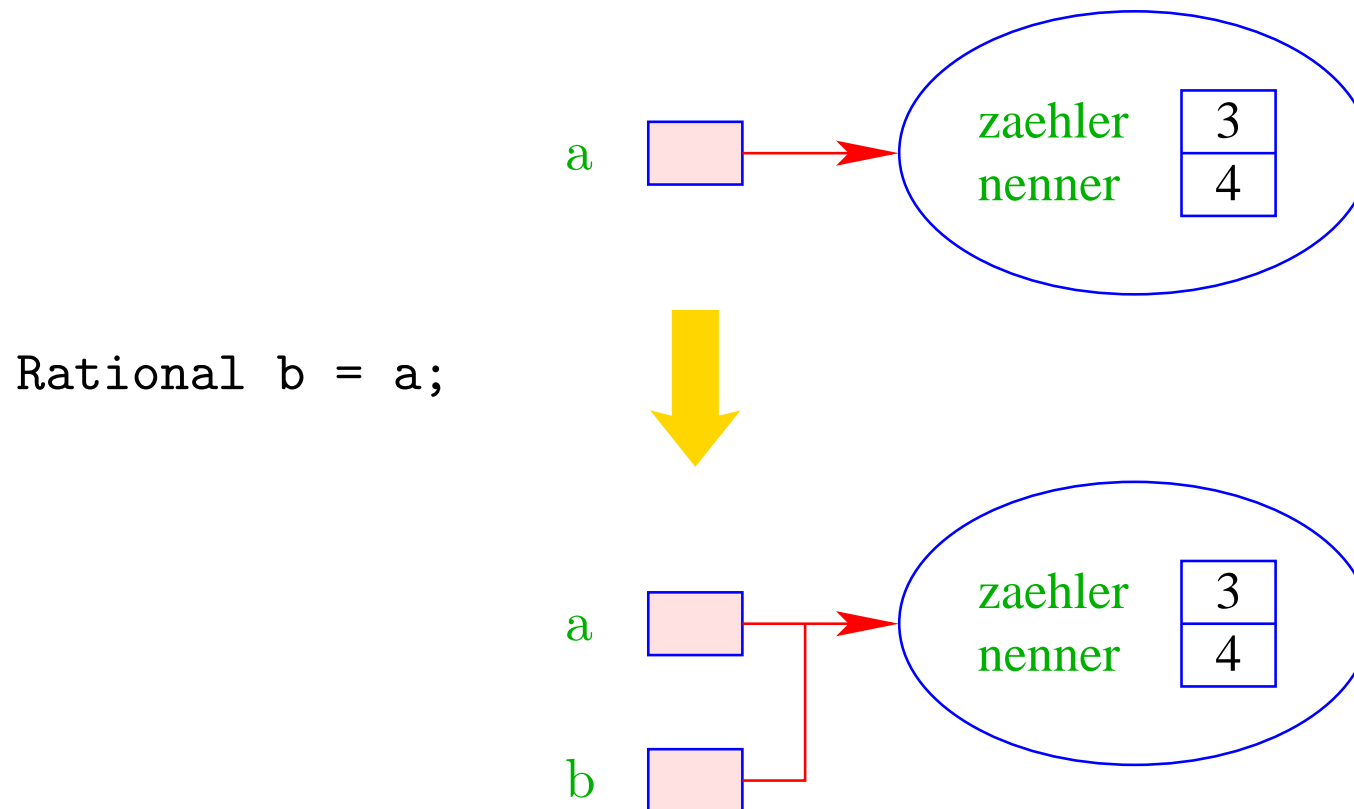
- Die Daten-Komponenten eines Objekts heißen **Instanz-Variablen** oder **Attribute**.

- Rational **name** ; deklariert eine Variable für Objekte der Klasse Rational.
- Das Kommando `new Rational(...)` legt das Objekt an, ruft einen **Konstruktor** für dieses Objekt auf und liefert das neue Objekt zurück:

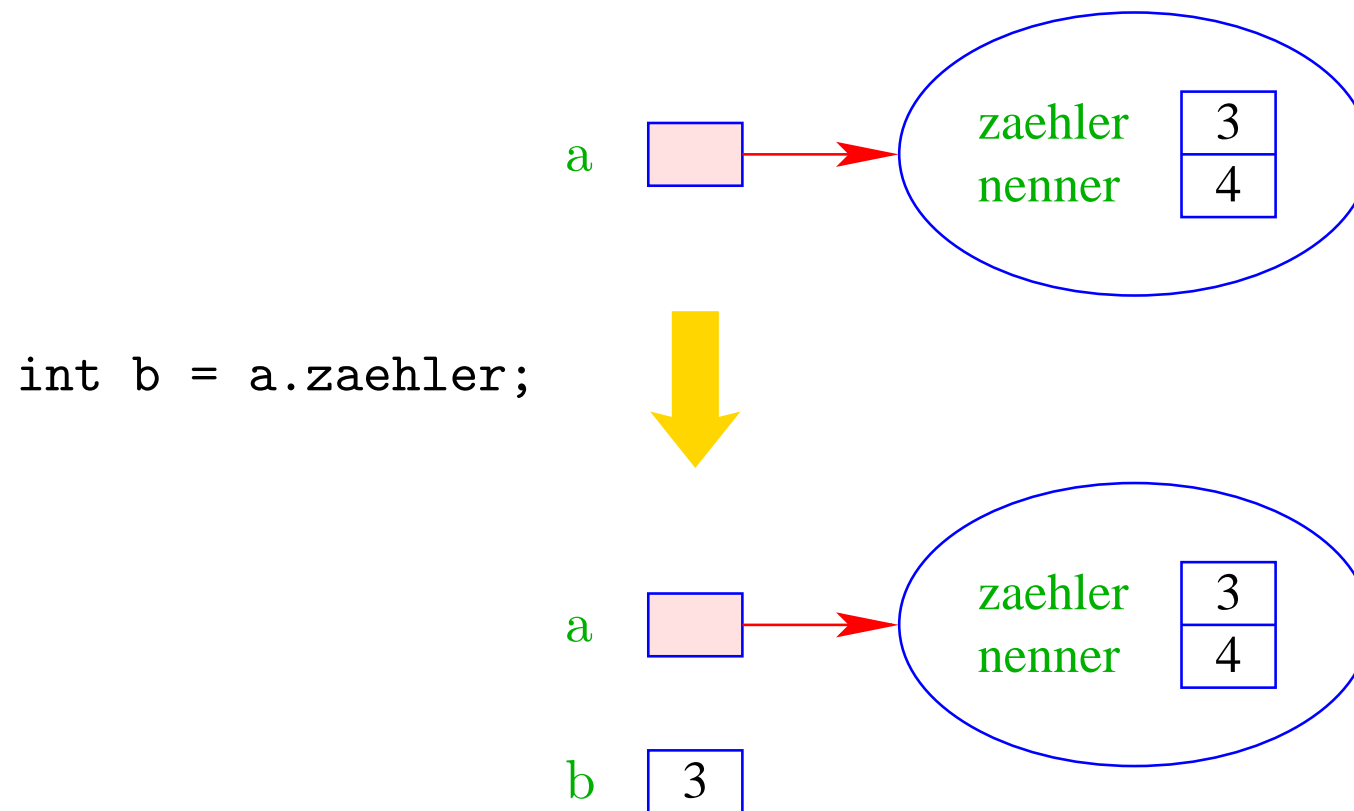


- Der Konstruktor ist eine Prozedur, die die Attribute des neuen Objekts initialisieren kann.

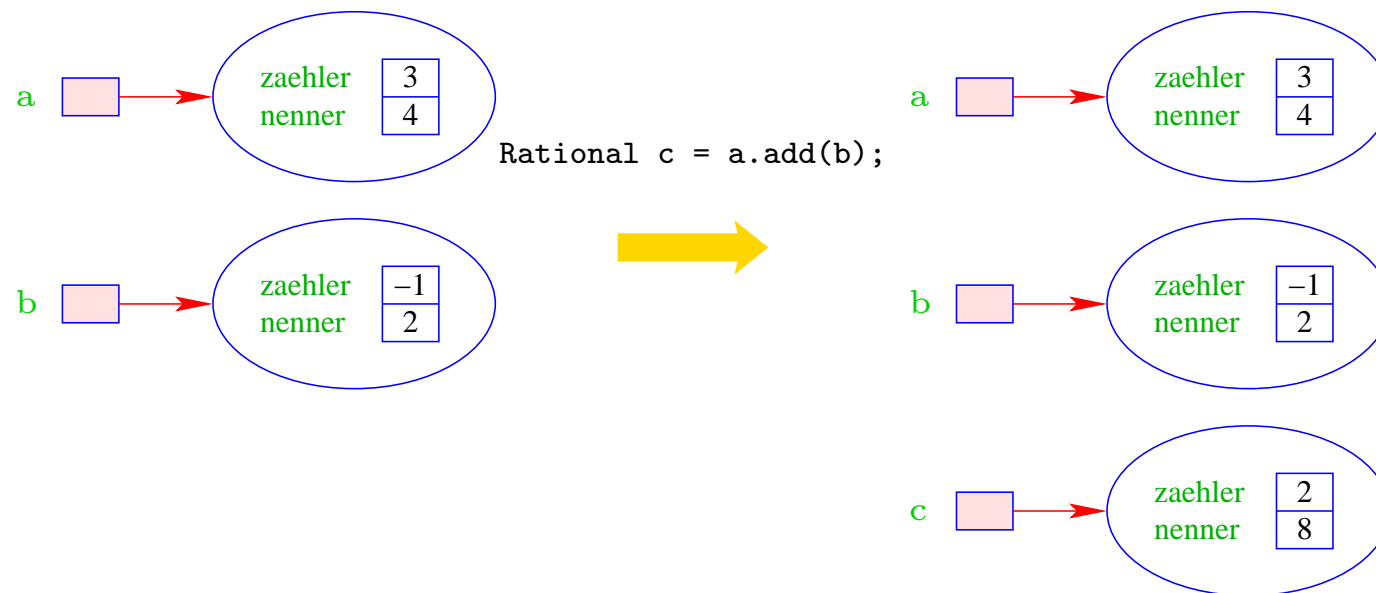
- Der Wert einer Rational-Variable ist ein **Verweis** auf einen Speicherbereich.
- `Rational b = a;` kopiert den Verweis aus `a` in die Variable `b`:

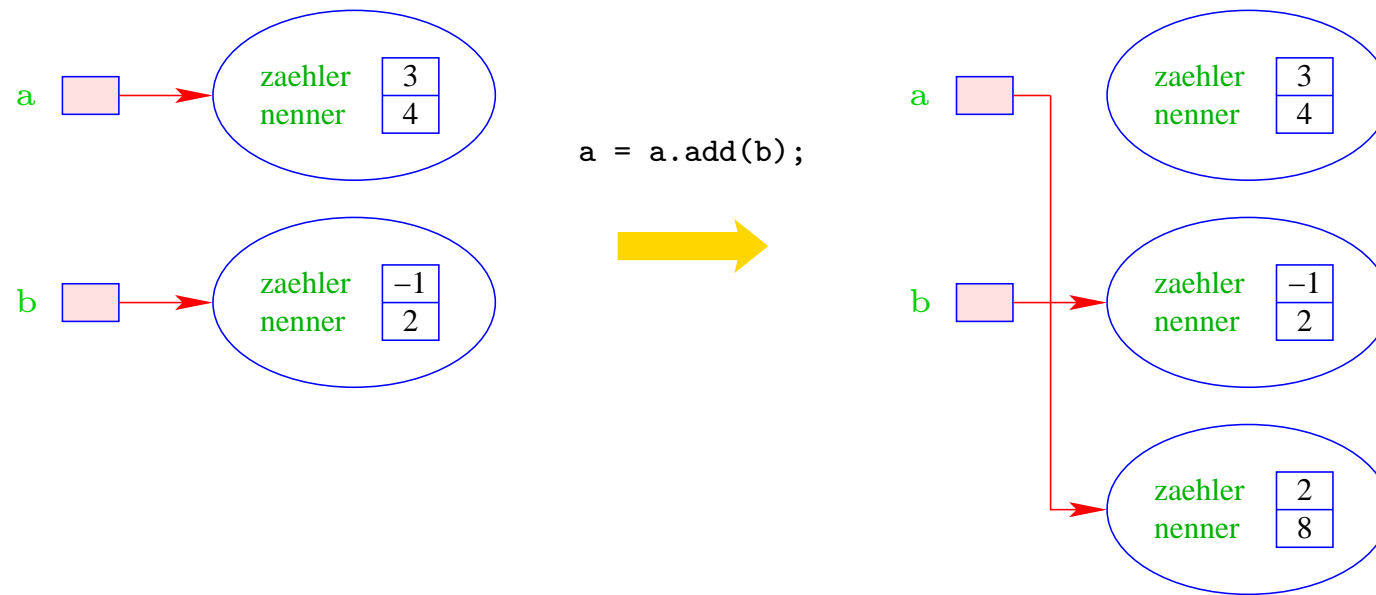


- `a.zaehler` liefert den Wert des Attributs `zaehler` des Objekts `a`:



- `a.add(b)` ruft die Operation `add` für `a` mit dem zusätzlichen aktuellen Parameter `b` auf:





- Die Operationen auf Objekten einer Klasse heißen auch **Methoden**, genauer: **Objekt-Methoden**.

## Zusammenfassung:

Eine Klassen-Deklaration besteht folglich aus Deklarationen von:

- **Attributen** für die verschiedenen Wert-Komponenten der Objekte;
- **Konstruktoren** zur Initialisierung der Objekte;
- **Methoden**, d.h. Operationen auf Objekten.

Diese Elemente heißen auch **Members** der Klasse.



```
public class Rational {  
    // Attribute:  
    private int zaehler, nenner;  
    // Konstruktoren:  
    public Rational (int x, int y) {  
        zaehler = x;  
        nenner = y;  
    }  
    public Rational (int x) {  
        zaehler = x;  
        nenner = 1;  
    }  
    ...  
}
```

```

// Objekt-Methoden:
public Rational add (Rational r) {
    int x = zaehler * r.nenner + r.zaehler * nenner;
    int y = nenner * r.nenner;
    return new Rational (x,y);
}

public boolean equals (Rational r) {
    return (zaehler * r.nenner == r.zaehler * nenner);
}

public String toString() {
    if (nenner == 1) return "" + zaehler;
    if (nenner > 0) return zaehler + "/" + nenner;
    return (-zaehler) + "/" + (-nenner);
}
} // end of class Rational

```

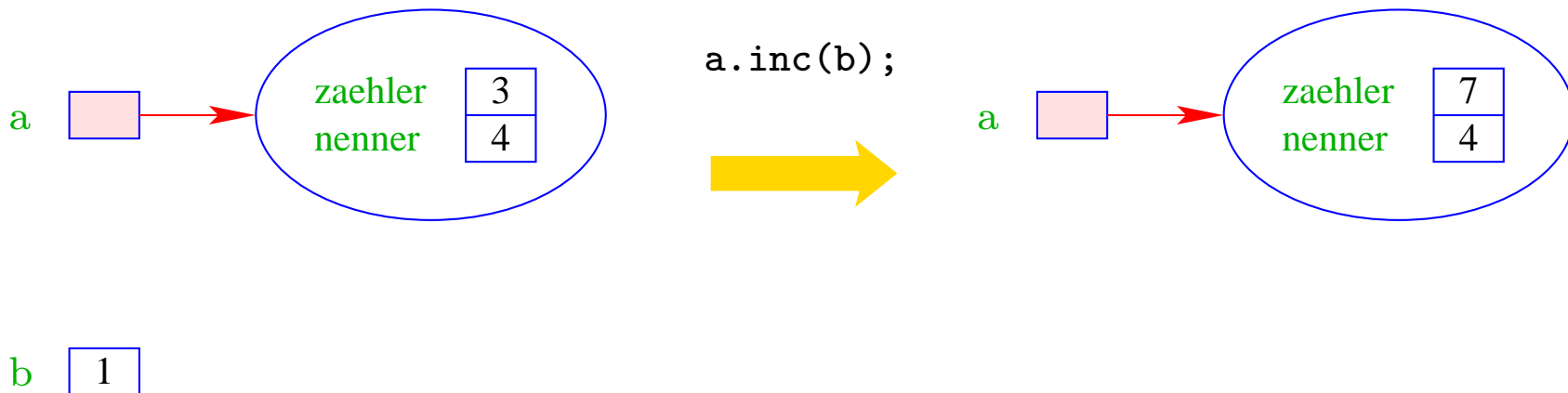
## Bemerkungen:

- Jede Klasse **sollte** in einer separaten Datei des entsprechenden Namens stehen.
- Die Schlüsselworte **private** bzw. **public** klassifizieren, für wen die entsprechenden Members sichtbar, d.h. zugänglich sind.
- **private** heißt: nur für Members der gleichen Klasse sichtbar.
- **public** heißt: innerhalb des gesamten Programms sichtbar.
- Nicht klassifizierte Members sind nur innerhalb des aktuellen **↑Package** sichtbar.

- Konstruktoren haben den gleichen Namen wie die Klasse.
- Es kann mehrere geben, sofern sie sich im Typ ihrer Argumente unterscheiden.
- Konstruktoren haben **keine** Rückgabewerte und darum auch keinen Rückgabebetyp.
- Methoden haben dagegen **stets** einen Rückgabe-Typ, evt. void.

```
public void inc (int b) {  
    zaehler = zaehler + b * nenner;  
}
```

- Die Objekt-Methode `inc()` modifiziert das Objekt, für das sie aufgerufen wurde.



- Die Objekt-Methode `equals()` ist nötig, da der Operator “==” bei Objekten die **Identität** der Objekte testet, d.h. die Gleichheit der Referenz !!!
- Die Objekt-Methode `toString()` liefert eine **String**-Darstellung des Objekts.
- Sie wird implizit aufgerufen, wenn das Objekt als Argument für die Konkatination “+” auftaucht.
- Innerhalb einer Objekt-Methode/eines Konstruktors kann auf die Attribute des Objekts **direkt** zugegriffen werden.
- **private**-Klassifizierung bezieht sich auf die Klasse nicht das Objekt: die Attribute **aller** `Rational`-Objekte sind für `add` sichtbar !!

Eine graphische Visualisierung der Klasse **Rational**, die nur die wesentliche Funktionalität berücksichtigt, könnte so aussehen:

<b>Rational</b>	
–	zähler : int nenner : int
+	add (y : Rational) : Rational equals (y : Rational) : boolean toString () : String

## Diskussion und Ausblick:

- Solche Diagramme werden von der **UML**, d.h. der **Unified Modelling Language** bereitgestellt, um Software-Systeme zu entwerfen (↑**Software Engineering**)
- Für eine einzelne Klasse lohnt sich ein solches Diagramm nicht wirklich.
- Besteht ein System aber aus **sehr vielen** Klassen, kann man damit die **Beziehungen** zwischen verschiedenen Klassen verdeutlichen.



## Diskussion und Ausblick:

- Solche Diagramme werden von der **UML**, d.h. der **Unified Modelling Language** bereitgestellt, um Software-Systeme zu entwerfen (↑**Software Engineering**)
- Für eine einzelne Klasse lohnt sich ein solches Diagramm nicht wirklich.
- Besteht ein System aber aus **sehr vielen** Klassen, kann man damit die **Beziehungen** zwischen verschiedenen Klassen verdeutlichen.

## Achtung:

**UML** wurde nicht speziell für **Java** entwickelt. Darum werden Typen abweichend notiert. Auch lassen sich manche Ideen nicht oder nur schlecht modellieren.

## 9.1 Selbst-Referenzen

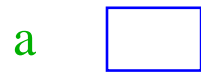
```
public class Cyclic {  
    private int info;  
    private Cyclic ref;  
    // Konstruktor  
    public Cyclic() {  
        info = 17;  
        ref = this;  
    }  
    ...  
} // end of class Cyclic
```

## 9.1 Selbst-Referenzen

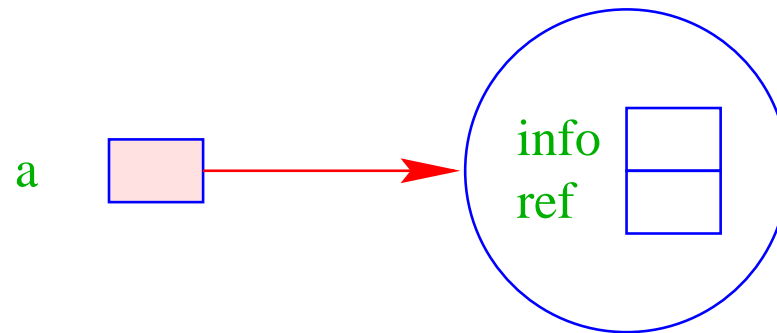
```
public class Cyclic {  
    private int info;  
    private Cyclic ref;  
    // Konstruktor  
    public Cyclic() {  
        info = 17;  
        ref = this;  
    }  
    ...  
} // end of class Cyclic
```

Innerhalb eines Members kann man mithilfe von `this` auf das aktuelle Objekt selbst zugreifen !

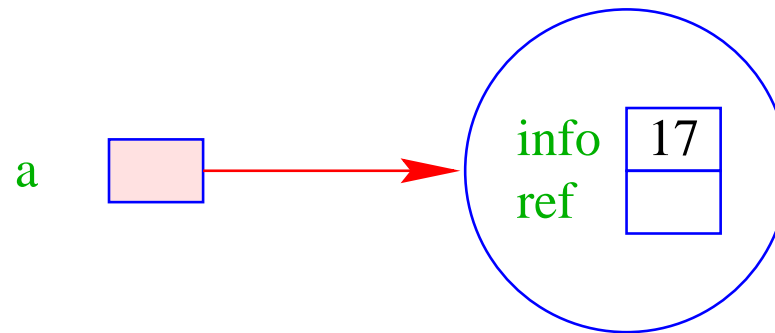
Für `Cyclic a = new Cyclic();` ergibt das:



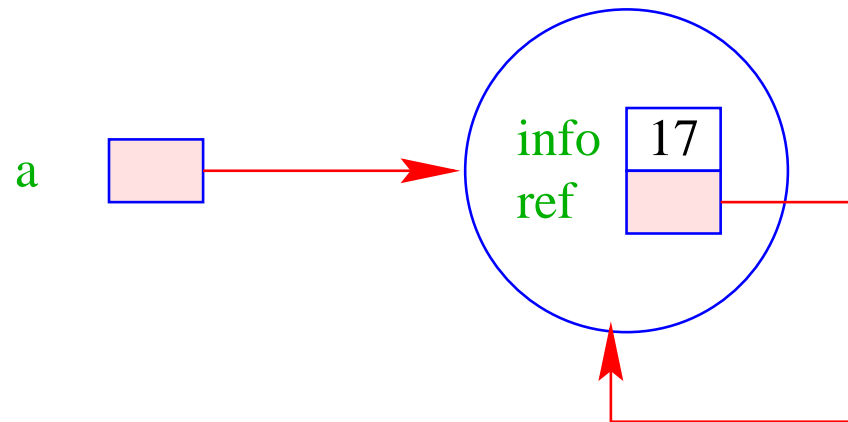
Für `Cyclic a = new Cyclic();` ergibt das:



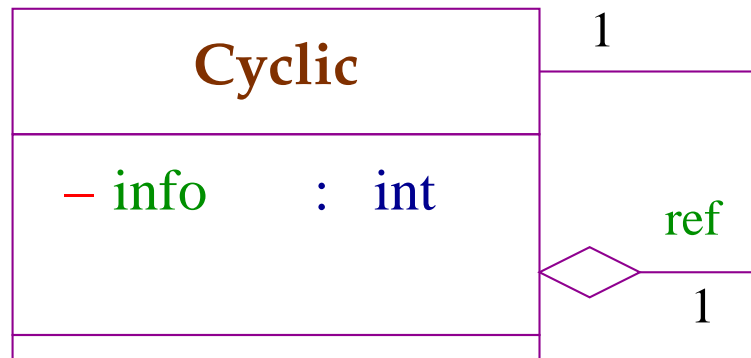
Für `Cyclic a = new Cyclic();` ergibt das:



Für `Cyclic a = new Cyclic();` ergibt das:



## Modellierung einer Selbst-Referenz:



Die Rauten-Verbindung heißt auch **Aggregation**.

Das Klassen-Diagramm vermerkt, dass jedes Objekt der Klasse **Cyclic** **einen** Verweis mit dem Namen **ref** auf **ein** weiteres Objekt der Klasse **Cyclic** enthält.



## 9.2 Klassen-Attribute

- Objekt-Attribute werden für jedes Objekt neu angelegt,
- **Klassen**-Attribute einmal für die gesamte Klasse.
- Klassen-Attribute erhalten die Qualifizierung `static`.

```
public class Count {  
    private static int count = 0;  
    private int info;  
    // Konstruktor  
    public Count() {  
        info = count; count++;  
    } ...  
} // end of class Count
```

count

0

```
Count a = new Count();
```



count

1

a



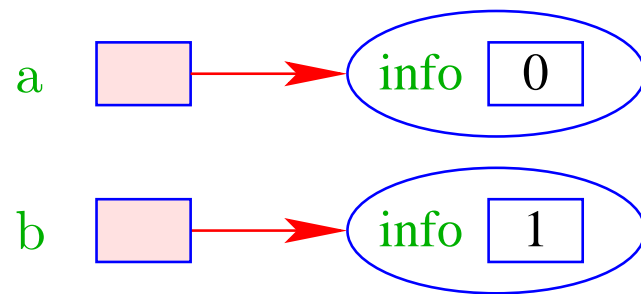
info

0

```
Count b = new Count();
```



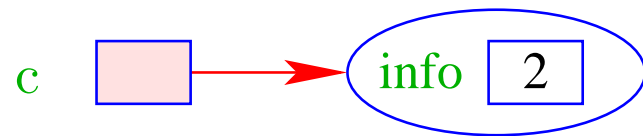
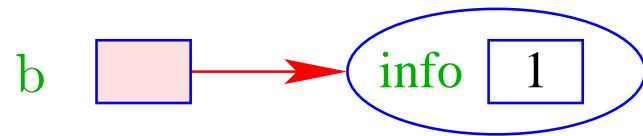
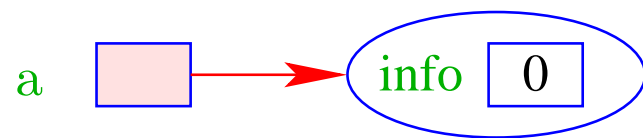
count 2



```
Count c = new Count();
```



count      3



- Das Klassen-Attribut `count` zählt hier die Anzahl der bereits erzeugten Objekte.
- Das Objekt-Attribut `info` enthält für jedes Objekt eine eindeutige Nummer.
- Außerhalb der Klasse `Class` kann man auf eine öffentliche Klassen-Variable `name` mithilfe von `Class.name` zugreifen.

- Das Klassen-Attribut `count` zählt hier die Anzahl der bereits erzeugten Objekte.
- Das Objekt-Attribut `info` enthält für jedes Objekt eine eindeutige Nummer.
- Außerhalb der Klasse `Class` kann man auf eine öffentliche Klassen-Variable `name` mithilfe von `Class.name` zugreifen.
- Objekt-Methoden werden stets mit einem Objekt aufgerufen ...
- dieses Objekt fungiert wie ein weiteres Argument !
- Funktionen und Prozeduren der Klasse `ohne` dieses implizite Argument heißen `Klassen`-Methoden und werden durch das Schlüsselwort `static` kenntlich gemacht.

In `Rational` könnten wir definieren:

```
public static Rational[] intToRationalArray(int[] a) {  
    Rational[] b = new Rational[a.length];  
    for(int i=0; i < a.length; ++i)  
        b[i] = new Rational (a[i]);  
    return b;  
}
```



In `Rational` könnten wir definieren:

```
public static Rational[] intToRationalArray(int[] a) {  
    Rational[] b = new Rational[a.length];  
    for(int i=0; i < a.length; ++i)  
        b[i] = new Rational (a[i]);  
    return b;  
}
```

- Die Funktion erzeugt für ein Feld von `int`'s ein entsprechendes Feld von `Rational`-Objekten.
- Außerhalb der Klasse `Class` kann die öffentliche Klassen-Methode `meth()` mithilfe von `Class.meth(...)` aufgerufen werden.

# 10 Abstrakte Datentypen

- Spezifiziere nur die Operationen!
- Verberge Details
  - der Datenstruktur;
  - der Implementierung der Operationen.

$\implies$  Information Hiding

## Sinn:

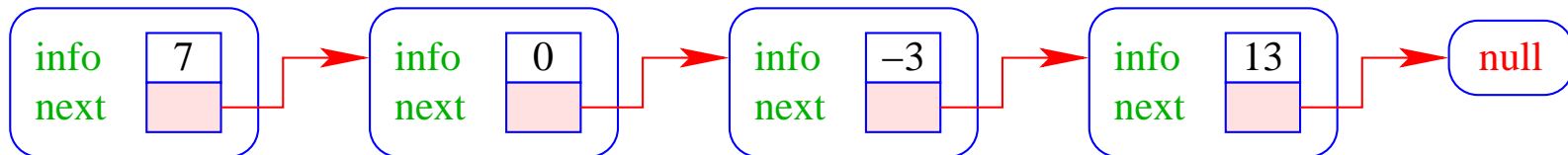
- Verhindern illegaler Zugriffe auf die Datenstruktur;
- **Entkopplung** von Teilproblemen für
  - Implementierung, aber auch
  - Fehlersuche und
  - Wartung;
- leichter **Austausch** von Implementierungen (**↑rapid prototyping**).

## 10.1 Ein konkreter Datentyp: Listen

Nachteil von Feldern:

- feste Größe;
- Einfügen neuer Elemente nicht möglich;
- Streichen ebenfalls nicht.

Idee: Listen



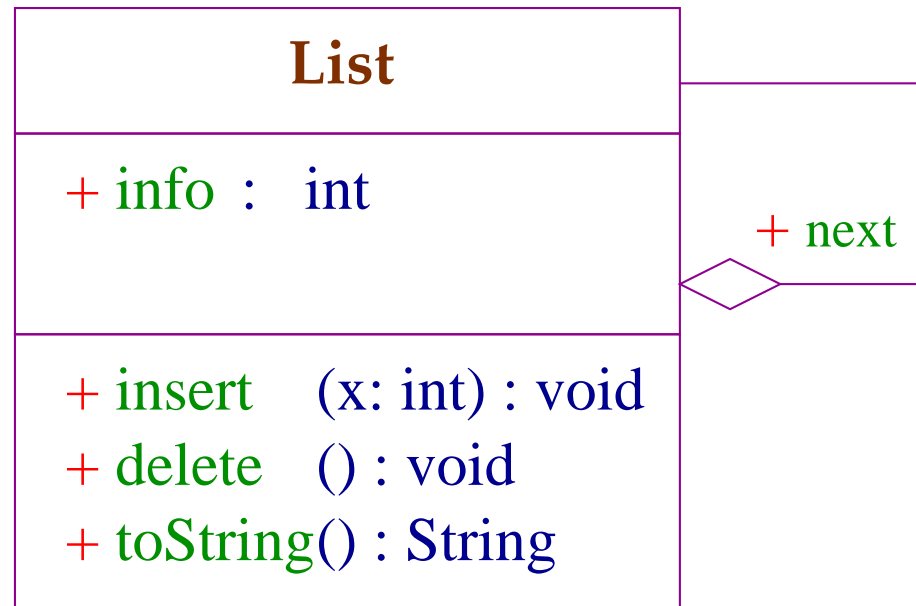
... das heißt:

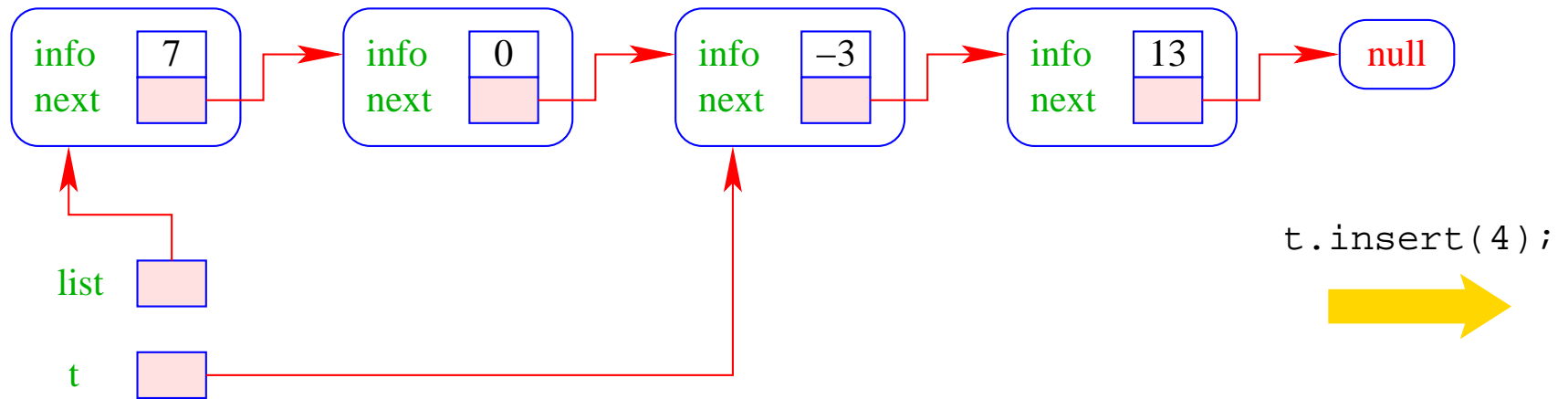
- `info` == Element der Liste;
- `next` == Verweis auf das nächste Element;
- `null` == leeres Objekt.

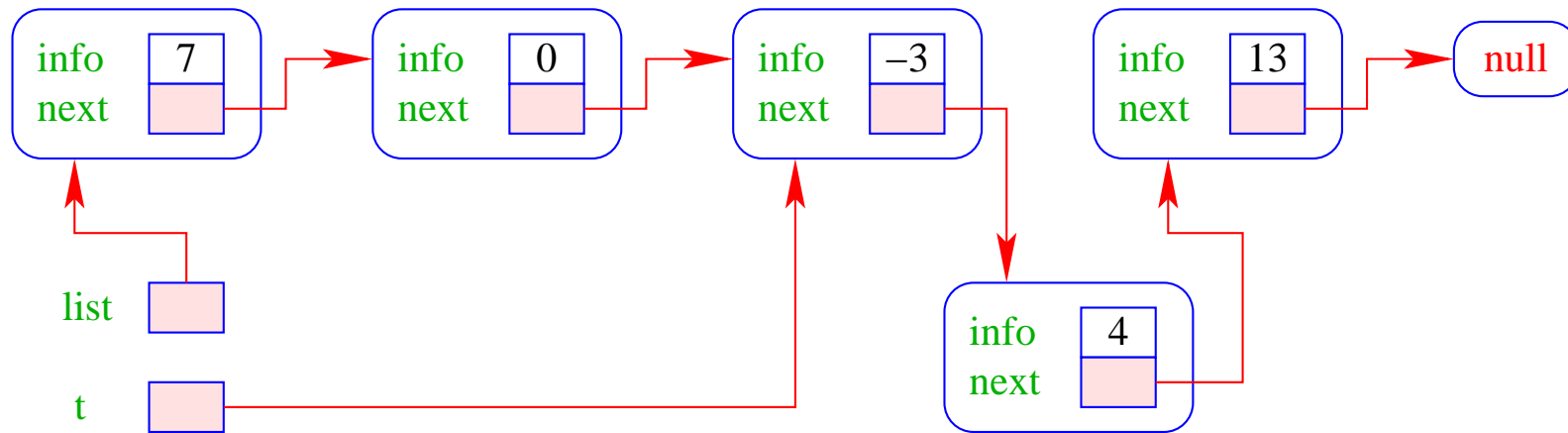
## Operationen:

`void insert(int x)` : fügt neues  $x$  hinter dem aktuellen Element ein;  
`void delete()` : entfernt Knoten hinter dem aktuellen Element;  
`String toString()` : liefert eine String-Darstellung.

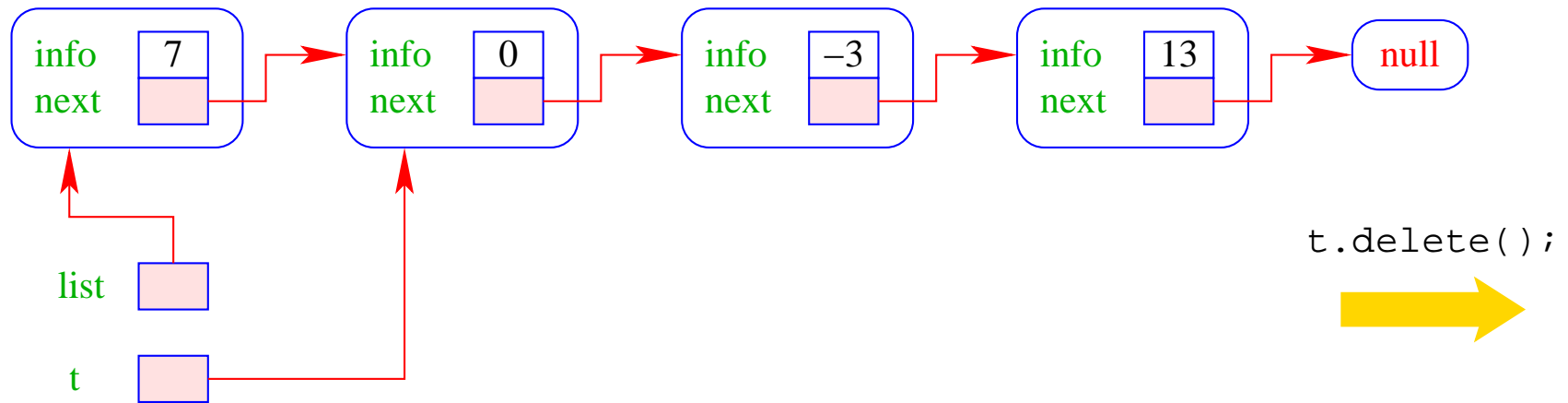
## Modellierung:

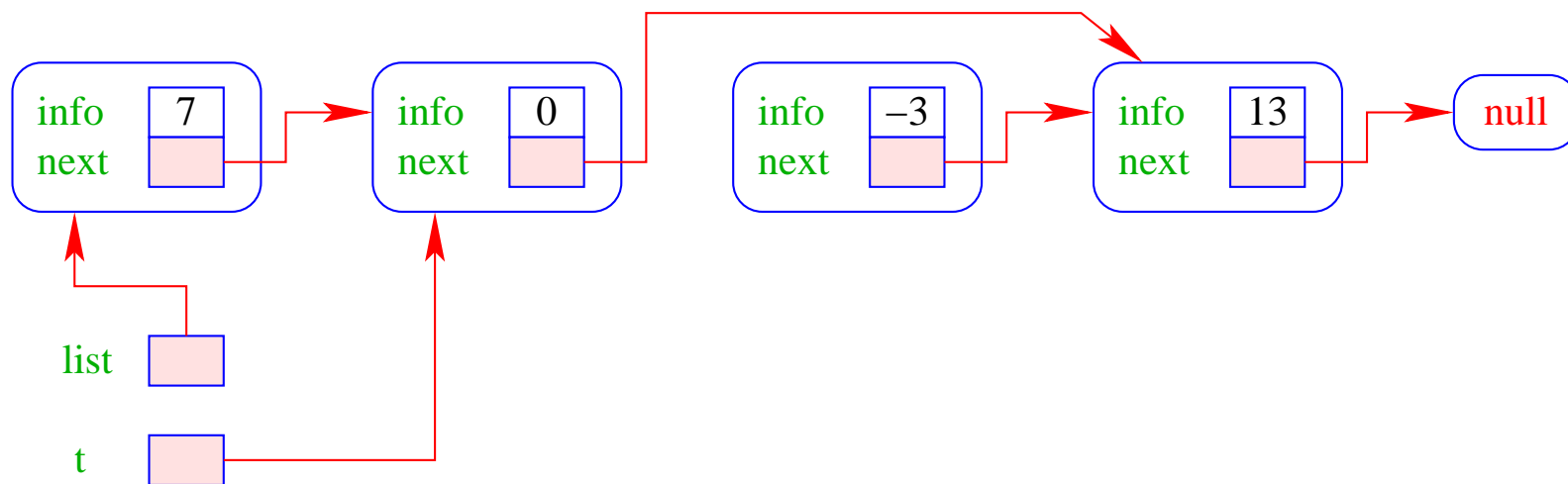












Weiterhin sollte man

- ... eine Liste auf Leerheit testen können;
- ... neue Listen bauen können, d.h. etwa:
  - ... eine ein-elementige Liste anlegen können;
  - ... eine Liste um ein Element verlängern können;
- ... Listen in Felder und Felder in Listen umwandeln können.

Weiterhin sollte man

- ... eine Liste auf Leerheit testen können;

**Achtung:**

das `null`-Objekt versteht **keinerlei** Objekt-Methoden!!!

- ... neue Listen bauen können, d.h. etwa:
  - ... eine ein-elementige Liste anlegen können;
  - ... eine Liste um ein Element verlängern können;
- ... Listen in Felder und Felder in Listen umwandeln können.

```
public class List {  
    public int info;  
    public List next;  
    // Konstruktoren:  
    public List (int x, List l) {  
        info = x;  
        next = l;  
    }  
    public List (int x) {  
        info = x;  
        next = null;  
    }  
    ...  
}
```

```
// Objekt-Methoden:
    public void insert(int x) {
        next = new List(x,next);
    }
    public void delete() {
        if (next != null)
            next = next.next;
    }
    public String toString() {
        String result = "["+info;
        for(List t=next; t!=null; t=t.next)
            result = result+", "+t.info;
        return result+"]";
    }
    ...

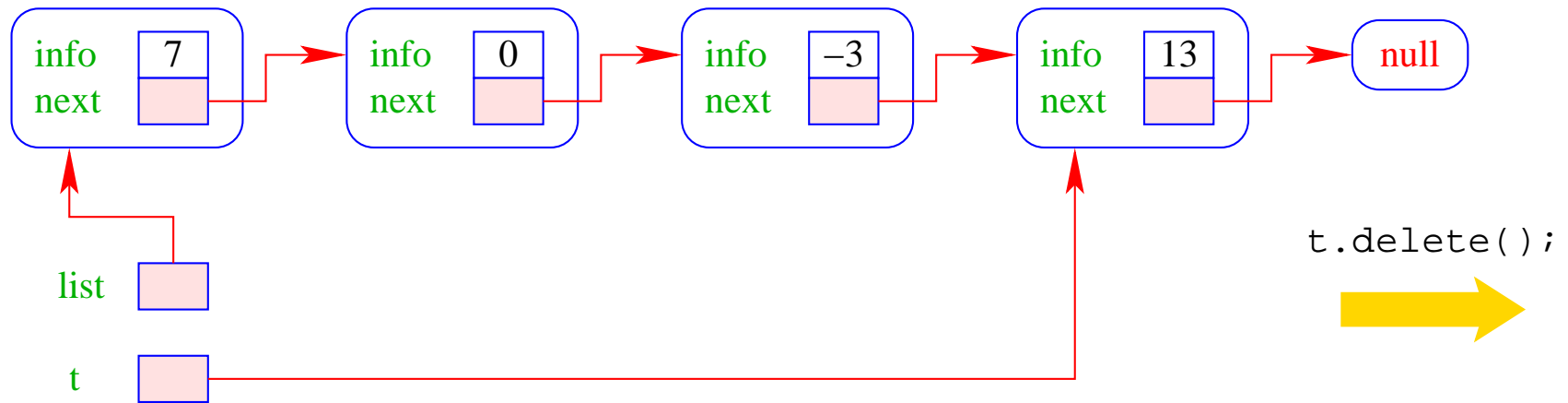
```

- Die Attribute sind `public` und daher beliebig einsehbar und modifizierbar  $\implies$  sehr flexibel, sehr fehleranfällig.
- `insert()` legt einen neuen Listenknoten an fügt ihn hinter dem aktuellen Knoten ein.
- `delete()` setzt den aktuellen `next`-Verweis auf das übernächste Element um.

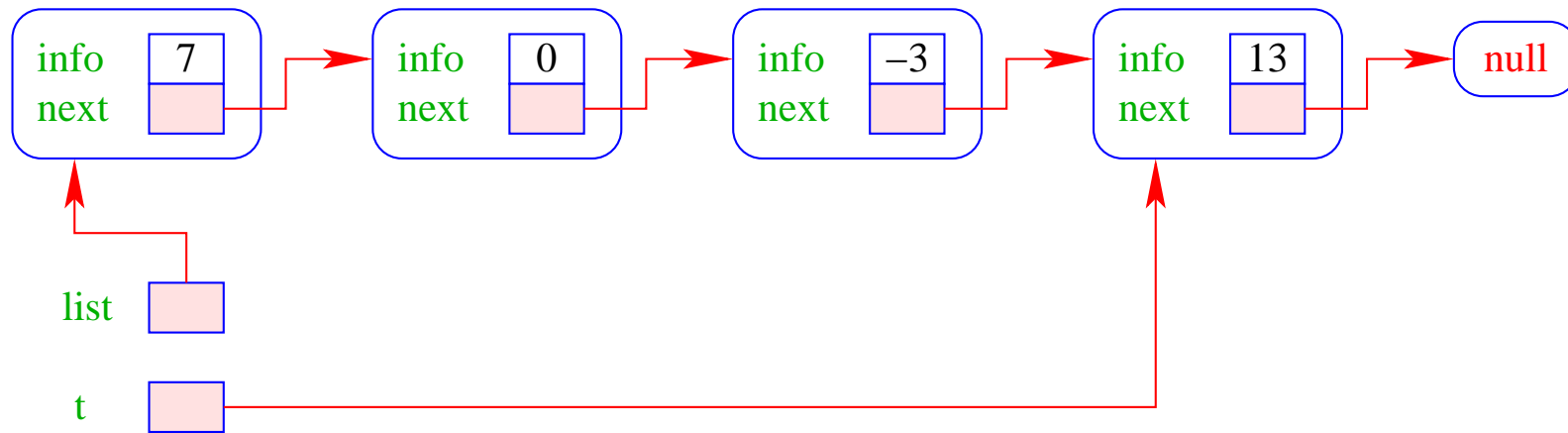
### Achtung:

Wenn `delete()` mit dem letzten Element der Liste aufgerufen wird, zeigt `next` auf `null`.

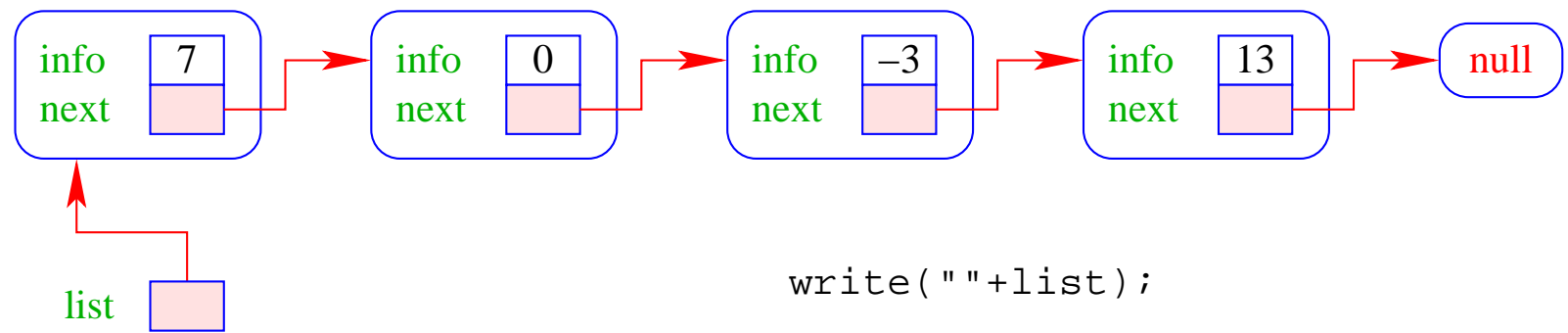
$\implies$  Wir tun dann nix.





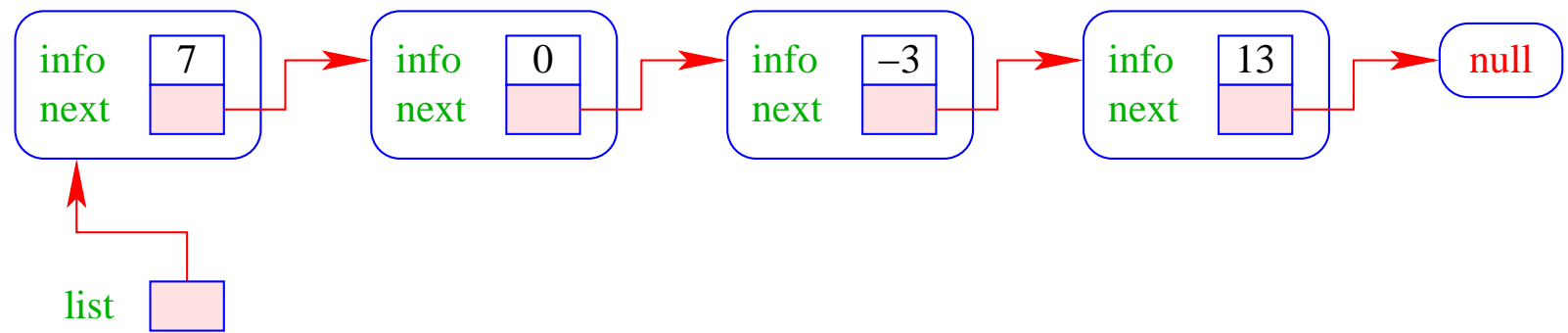


- Weil Objekt-Methoden nur für von `null` verschiedene Objekte aufgerufen werden können, kann die leere Liste nicht mittels `toString()` als `String` dargestellt werden.
- Der Konkatenations-Operator “+” ist so schlau, **vor** Aufruf von `toString()` zu überprüfen, ob ein `null`-Objekt vorliegt. Ist das der Fall, wird “null” ausgegeben.
- Wollen wir eine andere Darstellung, benötigen wir eine Klassen-Methode `String toString(List l)`.



`write(""+list);`





"[7, 0, -3, 13]"



```
write(""+list);
```





"null"

```
// Klassen-Methoden:
    public static boolean isEmpty(List l) {
        if (l == null)
            return true;
        else
            return false;
    }
    public static String toString(List l) {
        if (l == null)
            return "[]";
        else
            return l.toString();
    }
    ...
```

```

public static List arrayToList(int[] a) {
    List result = null;
    for(int i = a.length-1; i>=0; --i)
        result = new List(a[i],result);
    return result;
}

public int[] listToArray() {
    List t = this;
    int n = length();
    int[] a = new int[n];
    for(int i = 0; i < n; ++i) {
        a[i] = t.info;
        t = t.next;
    }
    return a;
}

```

...



- Damit das erste Element der Ergebnis-Liste `a[0]` enthält, beginnt die Iteration in `arrayToList()` beim **größten** Element.
- `listToArray()` ist als Objekt-Methode realisiert und funktioniert darum nur für **nicht-leere** Listen.
- Um eine Liste in ein Feld umzuwandeln, benötigen wir seine Länge.

```
private int length() {  
    int result = 1;  
    for(List t = next; t!=null; t=t.next)  
        result++;  
    return result;  
}  
} // end of class List
```

- Weil `length()` als `private` deklariert ist, kann es nur von den Methoden der Klasse `List` benutzt werden.
- Damit `length()` auch für `null` funktioniert, hätten wir analog zu `toString()` auch noch eine Klassen-Methode `int length(List l)` definieren können.
- Diese Klassen-Methode würde uns ermöglichen, auch eine Klassen-Methode `static int [] listToArray (List l)` zu definieren, die auch für leere Listen definiert ist.

**Anwendung:** Mergesort – Sortieren durch Mischen

**Mischen:**

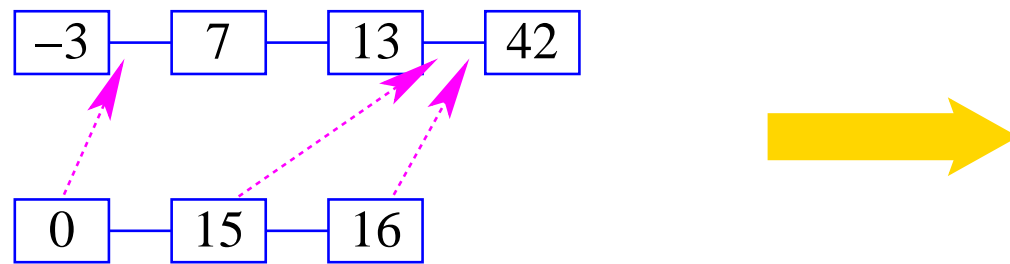
**Eingabe:** zwei sortierte Listen;

**Ausgabe:** eine gemeinsame sortierte Liste.

-3 — 7 — 13 — 42

0 — 15 — 16







## Idee:

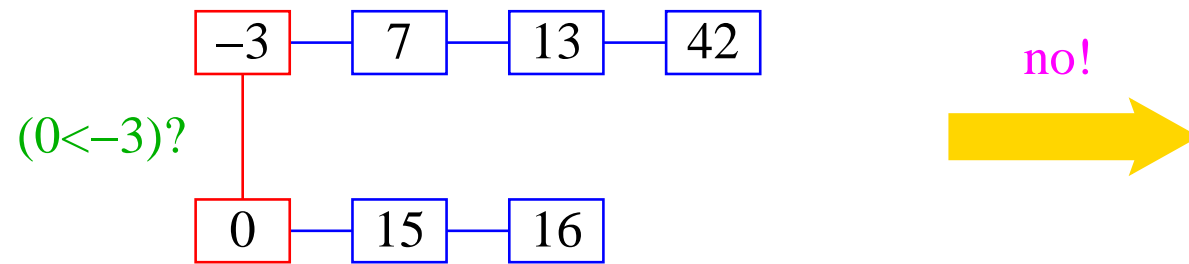
- Konstruiere sukzessive die Ausgabe-Liste aus den der Argument-Listen.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Input-Listen.
- Falls die  $n$  die Länge der längeren Liste ist, sind offenbar maximal nur  $n - 1$  Vergleiche nötig.

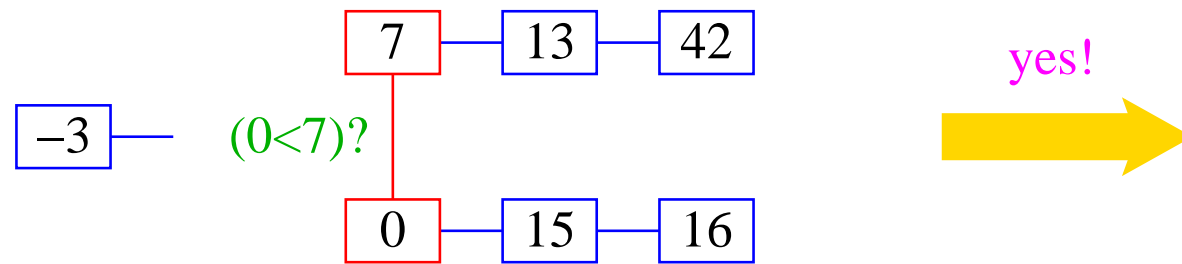
-3 — 7 — 13 — 42

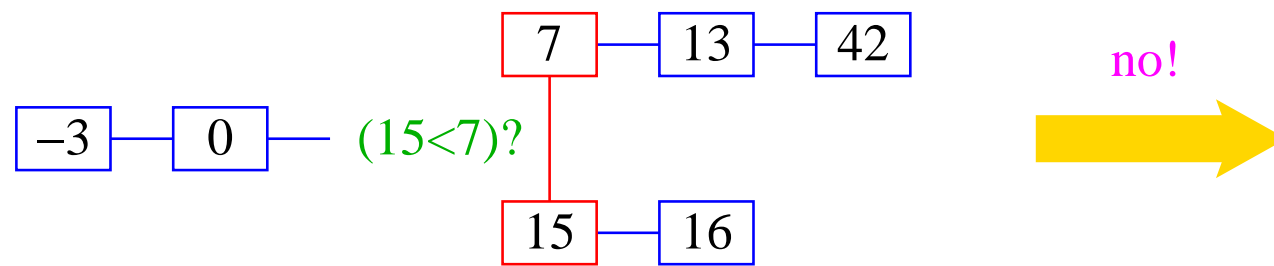
0 — 15 — 16

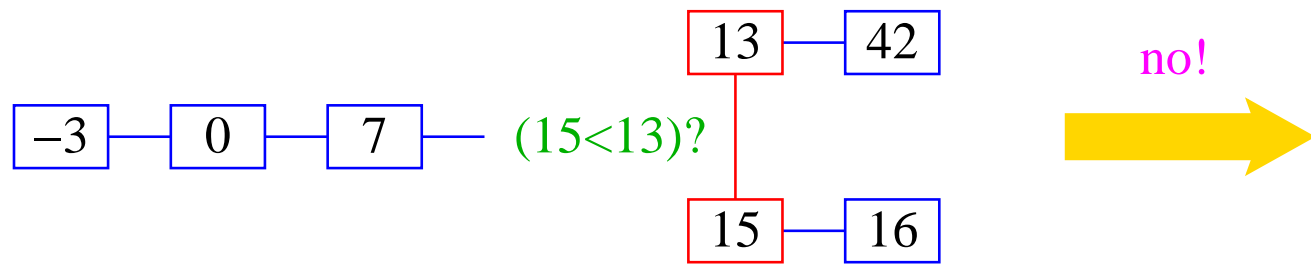


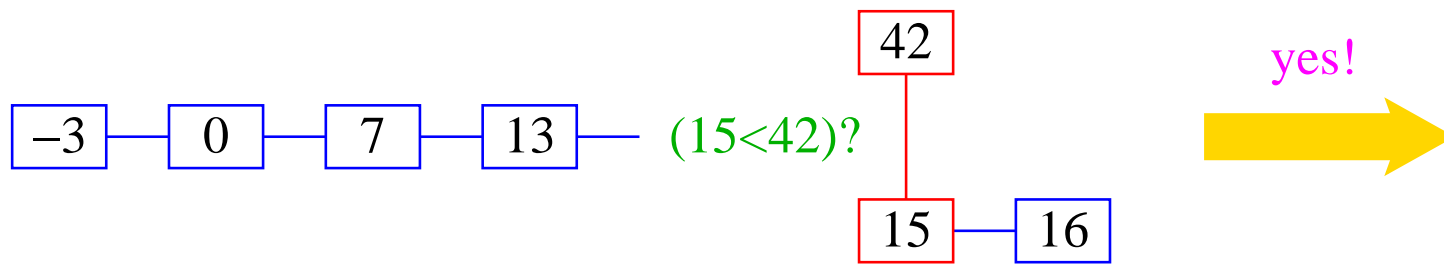


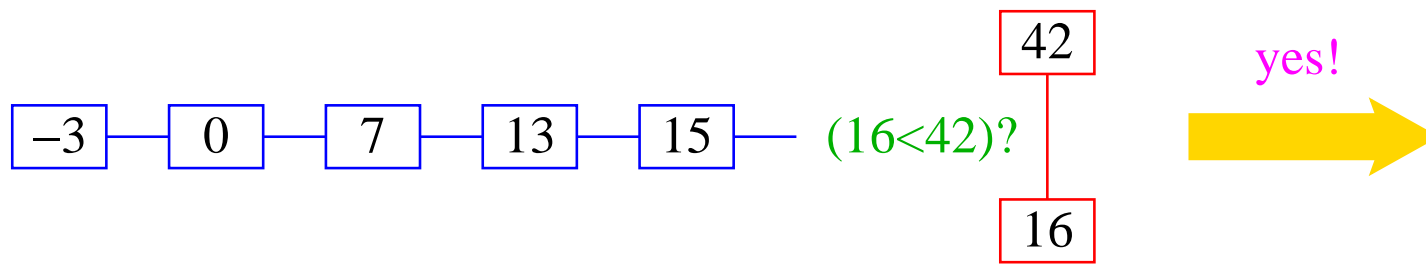












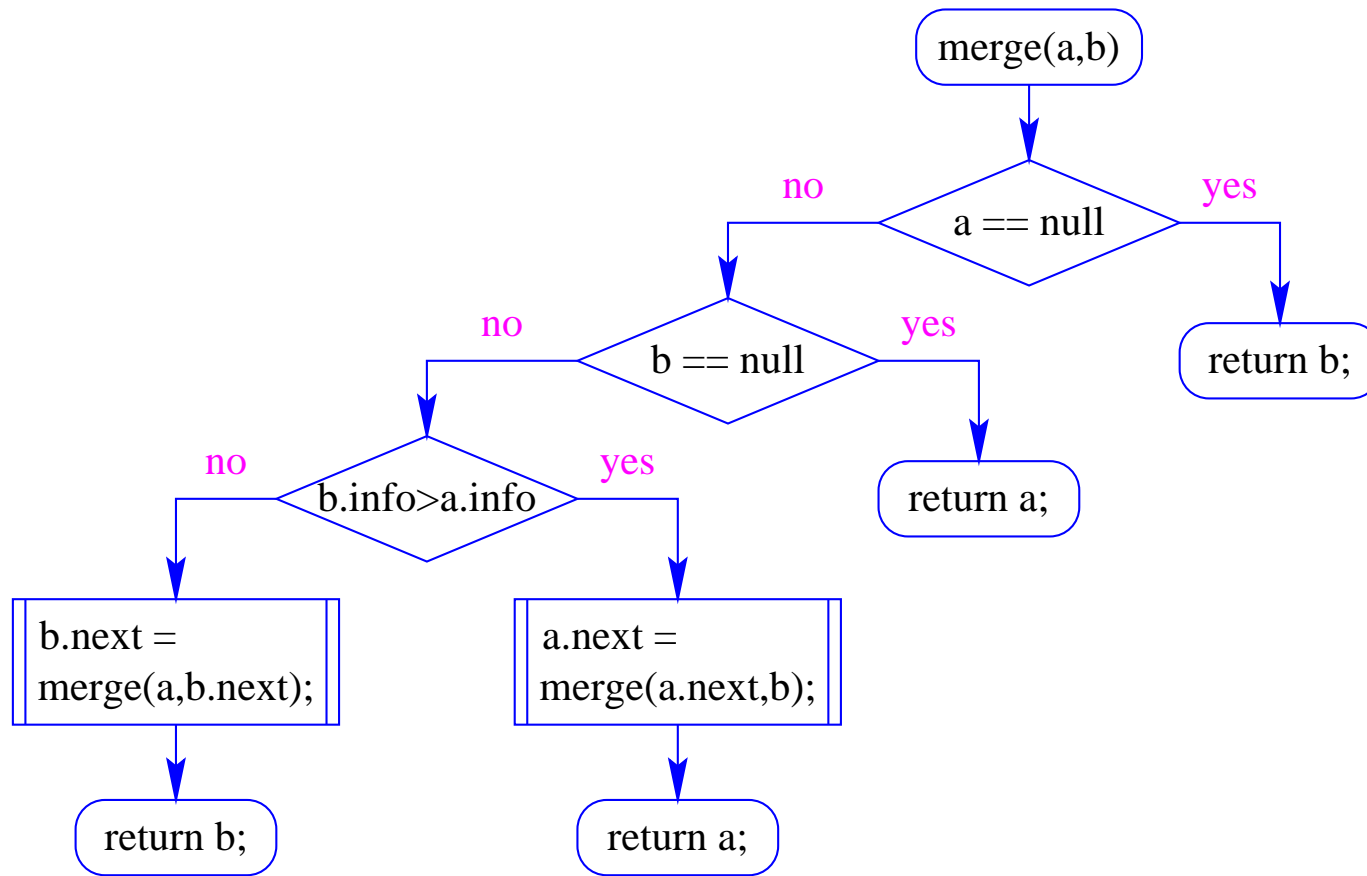


## Rekursive Implementierung:

- Falls eine der beiden Listen **a** und **b** leer ist, geben wir die andere aus.
- Andernfalls gibt es in jeder der beiden Listen ein erstes (kleinstes) Element.
- Von diesen beiden Elementen nehmen wir ein kleinstes.
- Dahinter hängen wir die Liste, die wir durch Mischen der verbleibenden Elemente erhalten ...

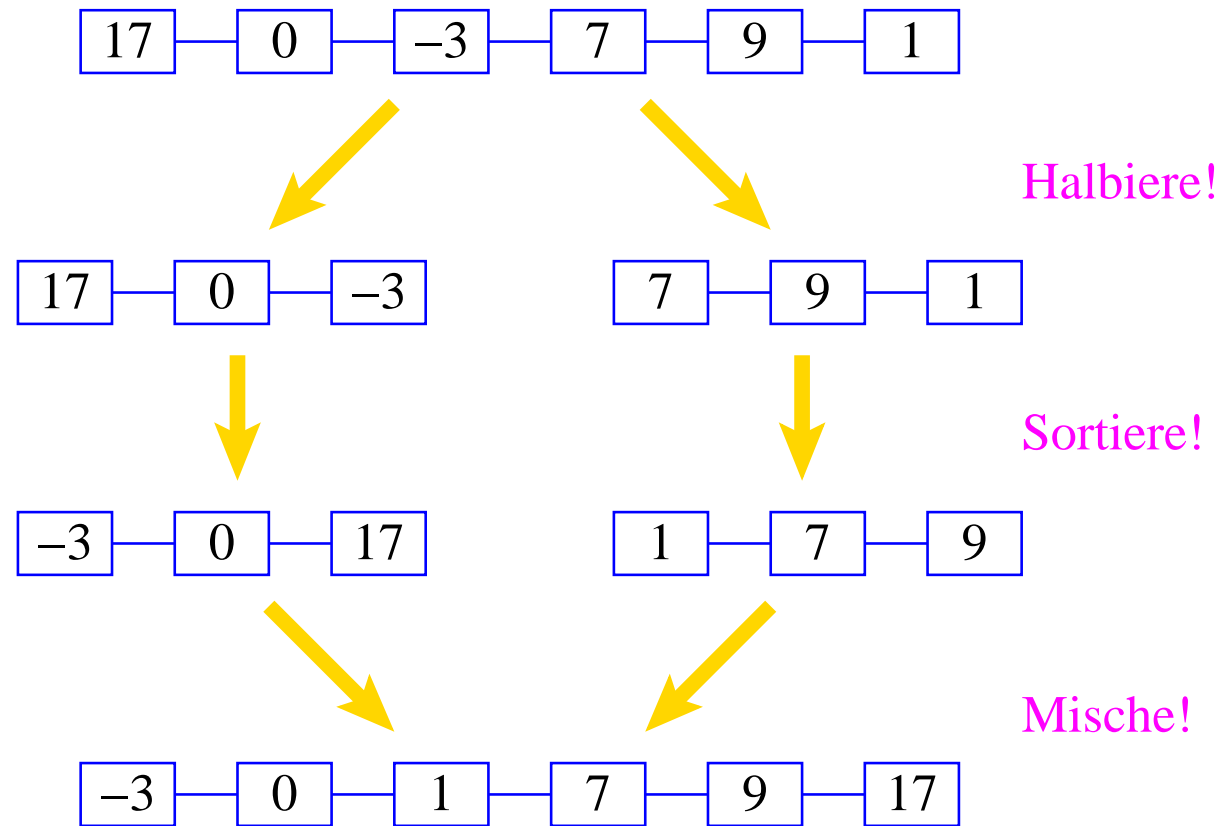


```
public static List merge(List a, List b) {  
    if (b == null)  
        return a;  
    if (a == null)  
        return b;  
    if (b.info > a.info) {  
        a.next = merge(a.next, b);  
        return a;  
    } else {  
        b.next = merge(a, b.next);  
        return b;  
    }  
}
```



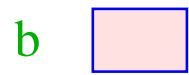
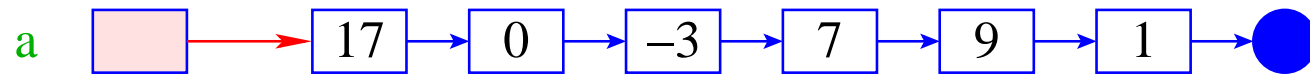
## Sortieren durch Mischen:

- Teile zu sortierende Liste in zwei Teil-Listen;
- sortiere jede Hälfte für sich;
- mische die Ergebnisse!



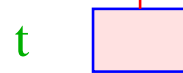
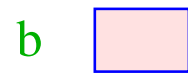
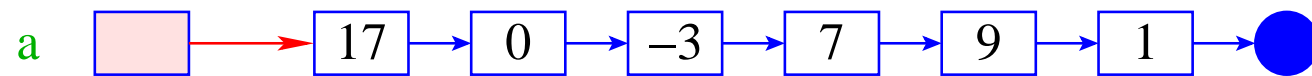
```
public static List sort(List a) {  
    if (a == null || a.next == null)  
        return a;  
    List b = a.half(); // Halbiere!  
    a = sort(a);  
    b = sort(b);  
    return merge(a,b);  
}
```

```
public List half() {  
    int n = length();  
    List t = this;  
    for(int i=0; i<n/2-1; i++)  
        t = t.next;  
    List result = t.next;  
    t.next = null;  
    return result;  
}
```

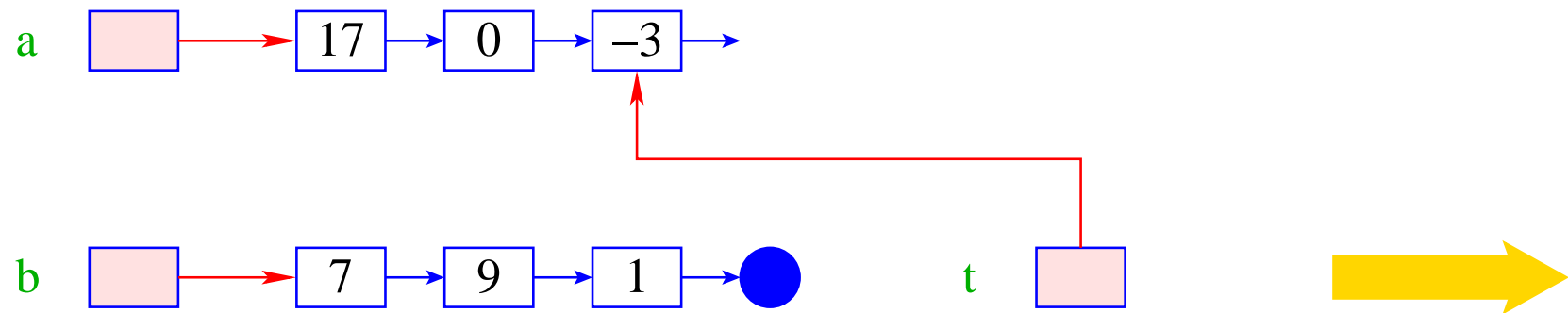


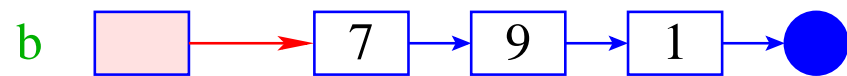
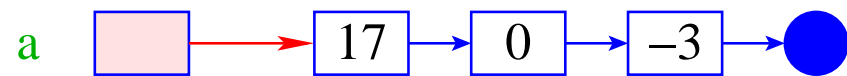
`b = a.half();`











## Diskussion:

- Sei  $V(n)$  die Anzahl der Vergleiche, die Mergesort maximal zum Sortieren einer Liste der Länge  $n$  benötigt.

Dann gilt:

$$V(1) = 0$$

$$V(2n) \leq 2 \cdot V(n) + 2 \cdot n$$

- Für  $n = 2^k$ , sind das dann nur  $k \cdot n$  Vergleiche !!!

## Achtung:

- Unsere Funktion `sort()` zerstört ihr Argument ?!
- Alle Listen-Knoten der Eingabe werden weiterverwendet.
- Die Idee des Sortierens durch Mischen könnte auch mithilfe von Feldern realisiert werden. (wie ?)
- Sowohl das Mischen wie das Sortieren könnte man statt rekursiv auch iterativ implementieren (wie ?)

## 10.2 Keller (Stacks)

### Operationen:

`boolean isEmpty()` : testet auf Leerheit;  
`int pop()` : liefert oberstes Element;  
`void push(int x)` : legt  $x$  oben auf dem Keller ab;  
`String toString()` : liefert eine String-Darstellung.

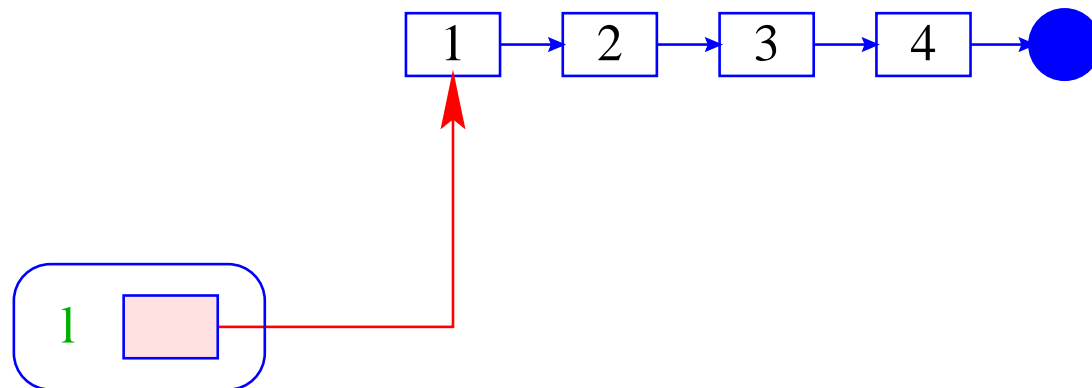
Weiterhin müssen wir einen leeren Keller anlegen können.

## Modellierung:

Stack	
+	Stack ()
+	isEmpty() : boolean
+	push (x: int) : void
+	pop () : int

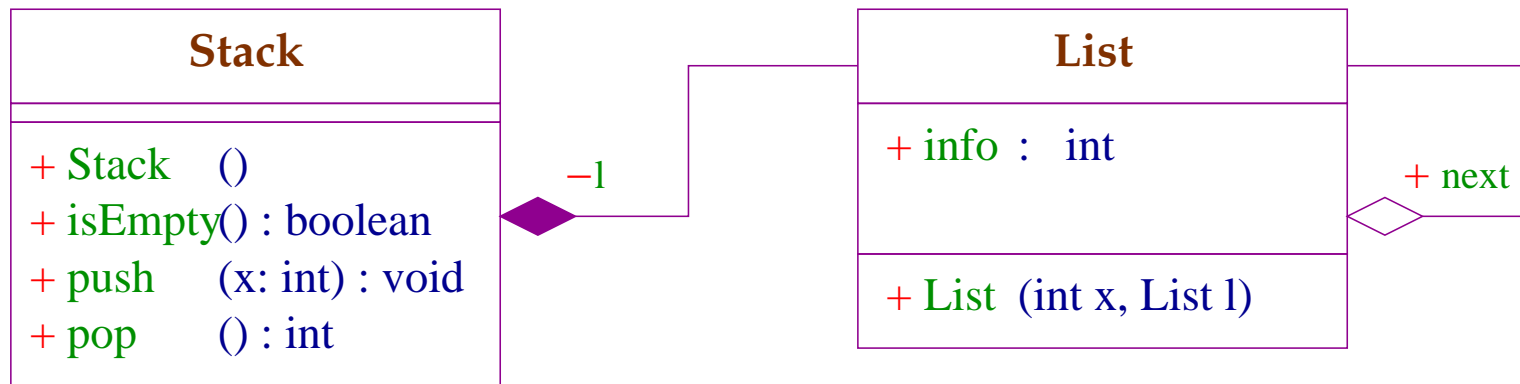
## Erste Idee:

- Realisiere Keller mithilfe einer Liste!



- Das Attribut 1 zeigt auf das oberste Element.

## Modellierung:



Die gefüllte Raute besagt, dass die Liste nur von Stack aus zugreifbar ist.



## Implementierung:

```
public class Stack {  
    private List l;  
    // Konstruktor:  
    public Stack() {  
        l = null;  
    }  
    // Objekt-Methoden:  
    public isEmpty() {  
        return List.isEmpty(l);  
    }  
    ...  
}
```

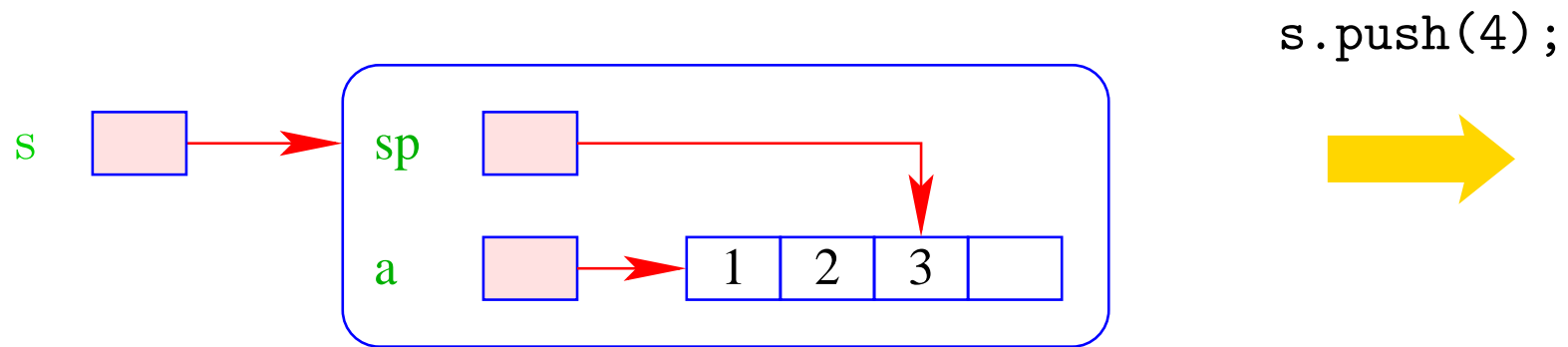
```
public int pop() {  
    int result = l.info;  
    l = l.next;  
    return result;  
}  
  
public void push(int a) {  
    l = new List(a,l);  
}  
  
public String toString() {  
    return List.toString(l);  
}  
  
} // end of class Stack
```

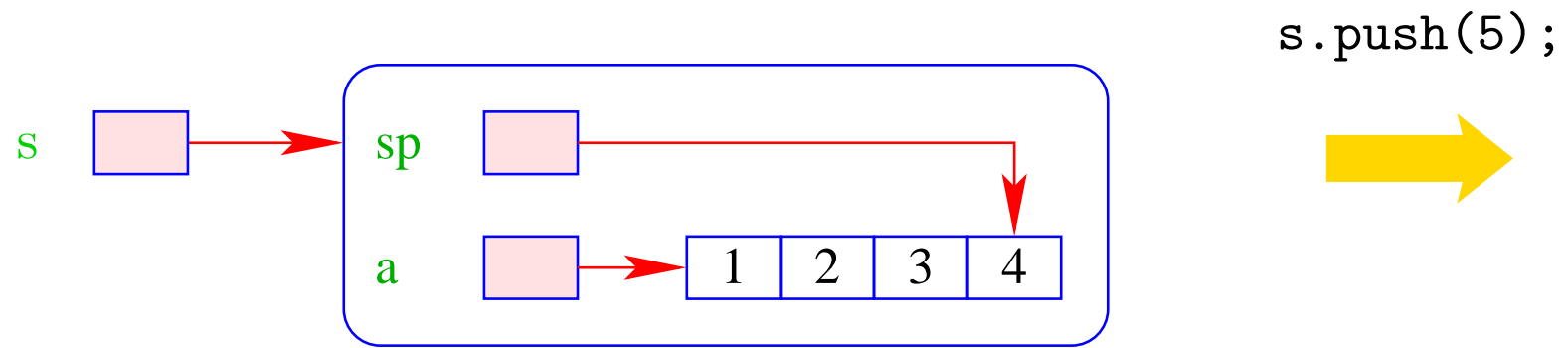
- Die Implementierung ist sehr einfach;
- ... nutzte gar nicht alle Features von `List` aus;
- ... die Listen-Elemente sind evt. über den gesamten Speicher verstreut;  
  
     $\implies$  führt zu schlechtem  $\uparrow$ Cache-Verhalten des Programms  
    !

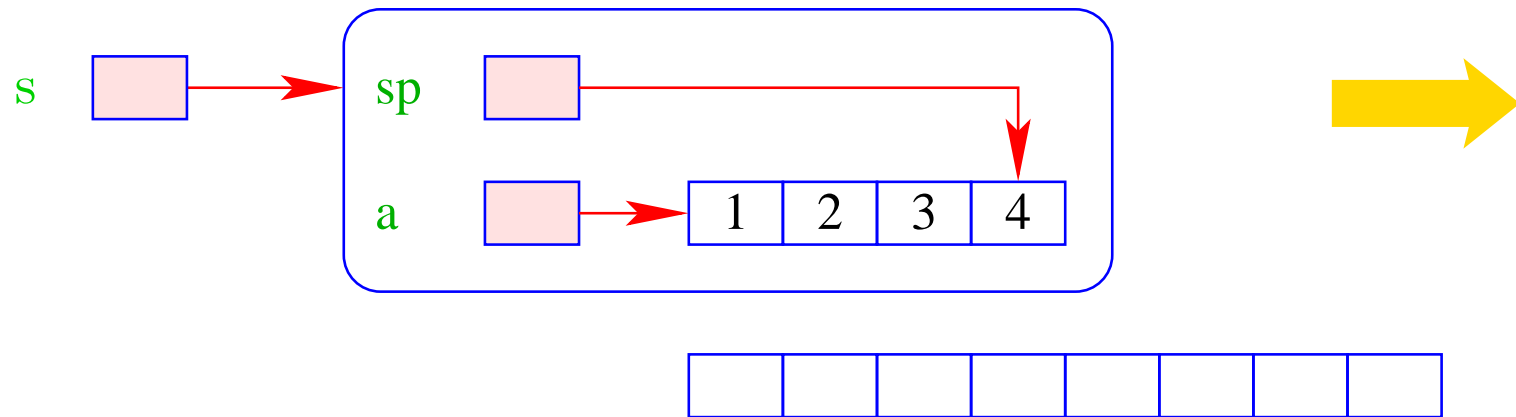
- Die Implementierung ist sehr einfach;
- ... nutzte gar nicht alle Features von `List` aus;
- ... die Listen-Elemente sind evt. über den gesamten Speicher verstreut;  
  
     $\implies$  führt zu schlechtem  $\uparrow$ Cache-Verhalten des Programms  
    !

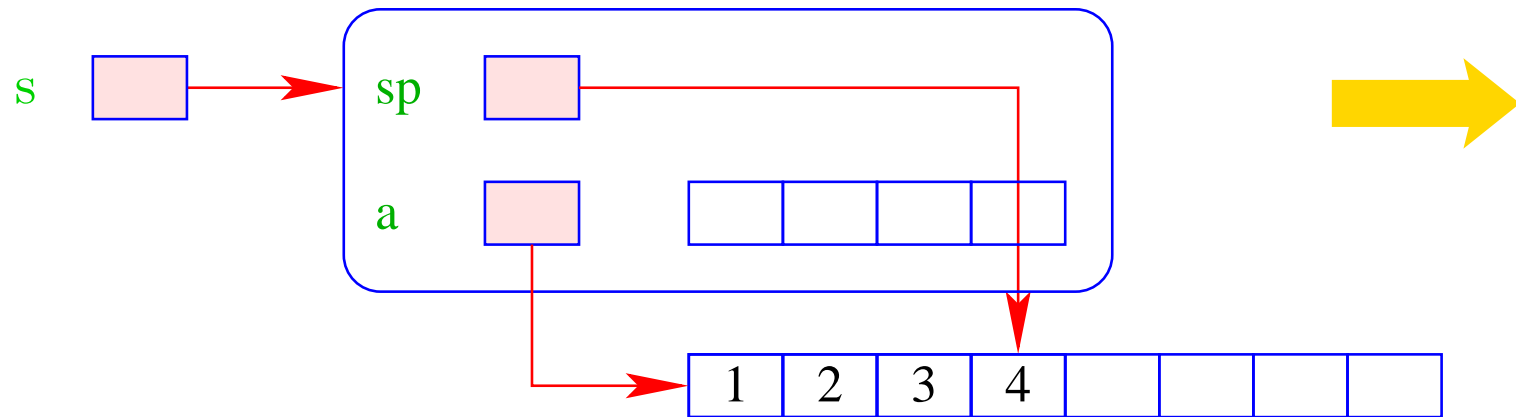
## Zweite Idee:

- Realisiere den Keller mithilfe eines Felds und eines Stackpointers, der auf die oberste belegte Zelle zeigt.
- Läuft das Feld über, ersetzen wir es durch ein größeres.

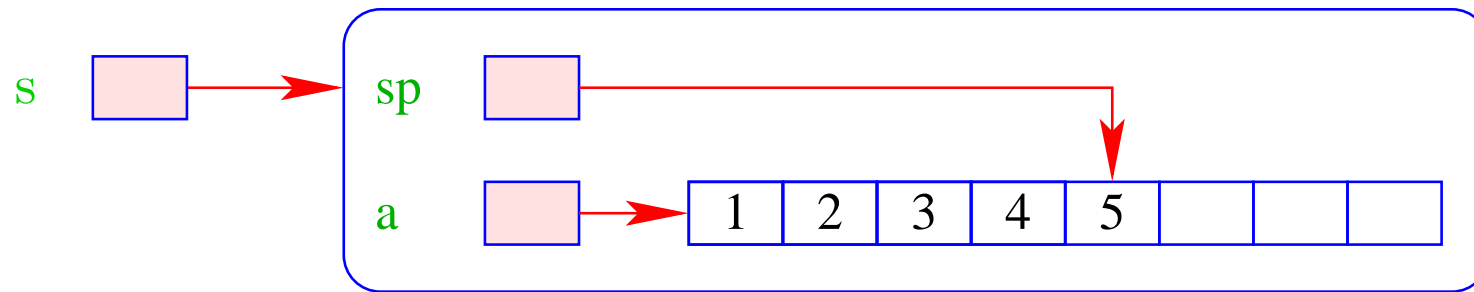




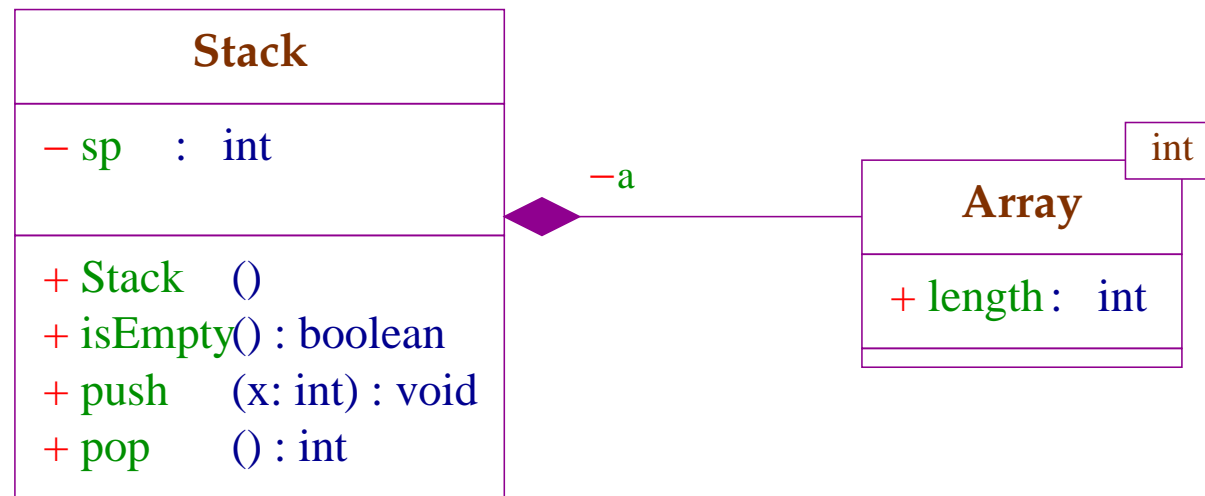








## Modellierung:



## Implementierung:

```
public class Stack {  
    private int sp;  
    private int[] a;  
    // Konstruktoren:  
    public Stack() {  
        sp = -1; a = new int[4];  
    }  
    // Objekt-Methoden:  
    public boolean isEmpty() {  
        return (sp<0);  
    }  
    ...  
}
```

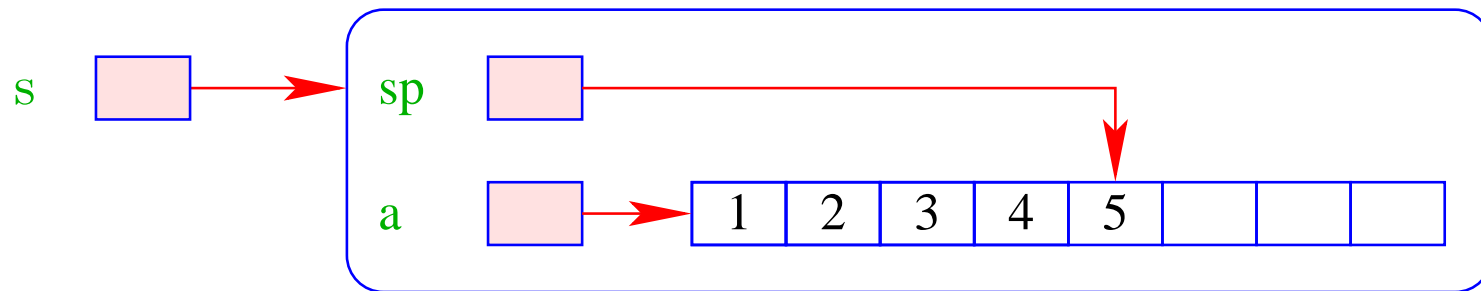
```
public int pop() {  
    return a[sp--];  
}  
public void push(int x) {  
    ++sp;  
    if (sp == a.length) {  
        int[] b = new int[2*sp];  
        for(int i=0; i<sp; ++i) b[i] = a[i];  
        a = b;  
    }  
    a[sp] = x;  
}  
public toString() {...}  
} // end of class Stack
```

## Nachteil:

- Es wird zwar neuer Platz allokiert, aber nie welcher freigegeben.

## Erste Idee:

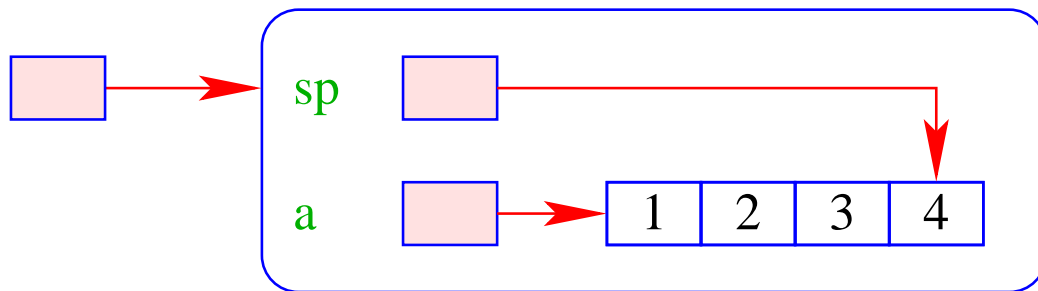
- Sinkt der Pegel wieder auf die Hälfte, geben wir diese frei ...



`x`

`x=s.pop( ) ;`

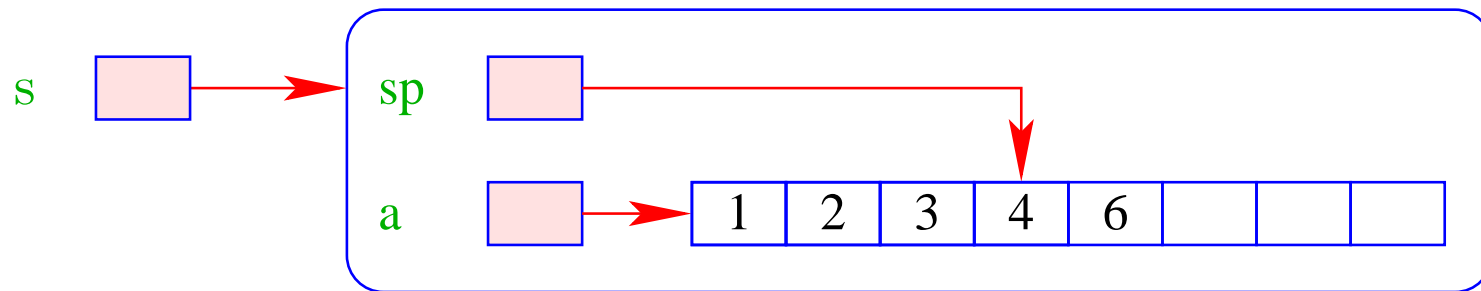




x 5

`s.push(6);`



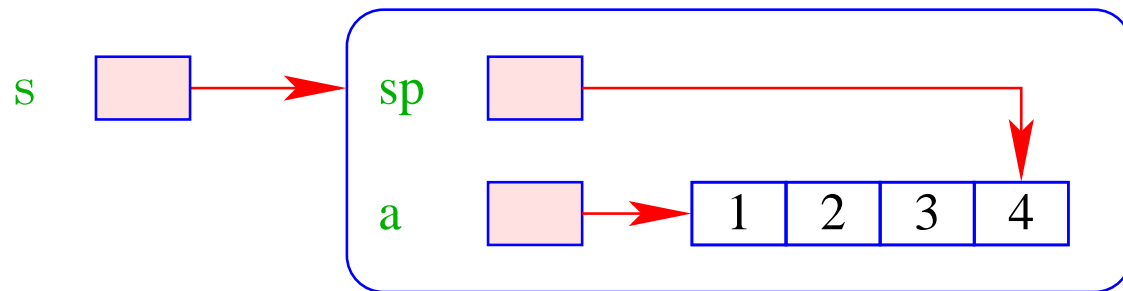


`x` 5

```
x = s.pop();
```



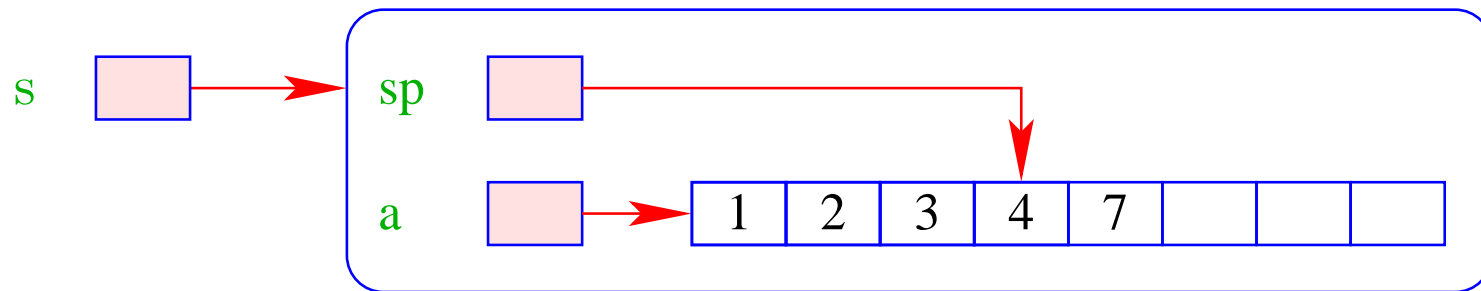





`x` 6

`s.push(7);`





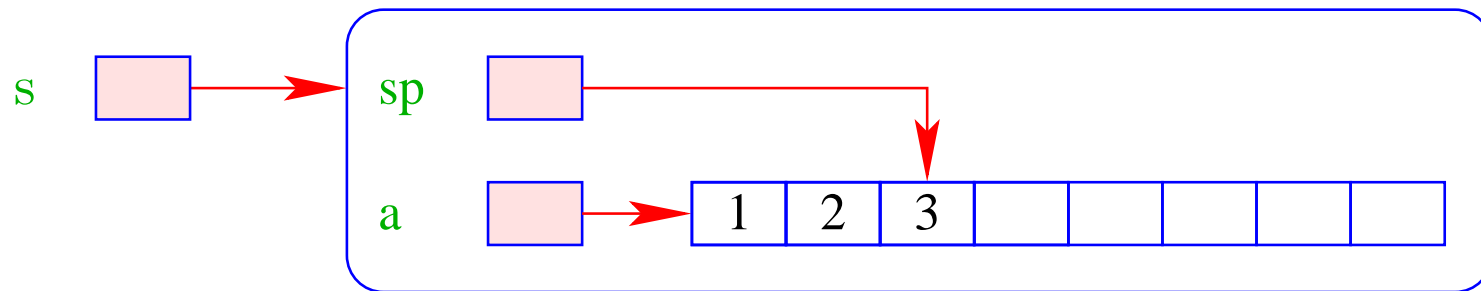
`x` 6

`x = s.pop( ) ;`  


- Im schlimmsten Fall müssen bei **jeder** Operation sämtliche Elemente kopiert werden.

## Zweite Idee:

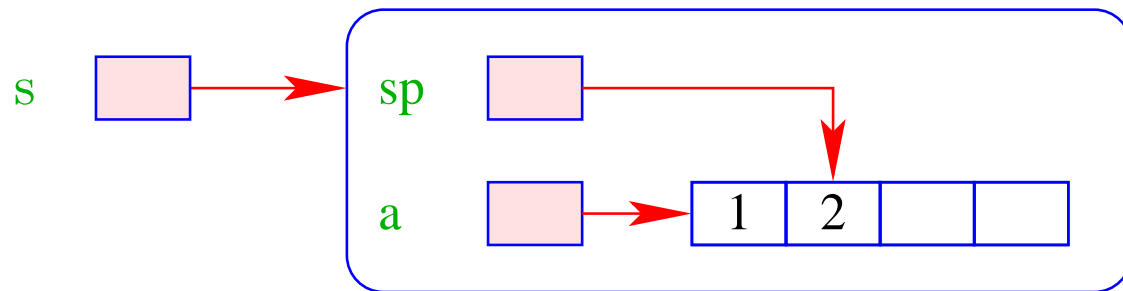
- Wir geben erst frei, wenn der Pegel auf **ein Viertel** fällt – und dann auch nur die Hälfte !



`x`

```
x = s.pop();
```

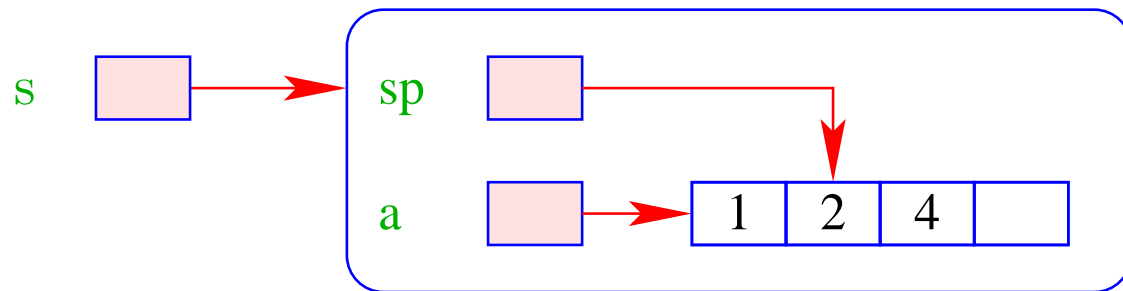




`x` 3

`s.push(4);`

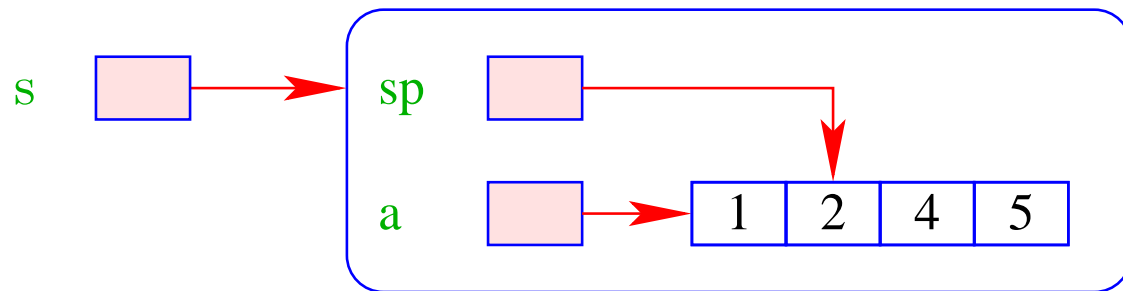




`x` 3

`s.push(5);`





$x$  3



- Vor jedem Kopieren werden **mindestens** halb so viele Operationen ausgeführt, wie Elemente kopiert werden.
- Gemittelt über die gesamte Folge von Operationen werden pro Operation maximal zwei Zahlen kopiert    ↑ **amortisierte Aufwandsanalyse**.



```
public int pop() {  
    int result = a[sp];  
    if (sp == a.length/4 && sp>=2) {  
        int[] b = new int[2*sp];  
        for(int i=0; i < sp; ++i)  
            b[i] = a[i];  
        a = b;  
    }  
    sp--;  
    return result;  
}
```

## 10.3 Schlangen (Queues)

(Warte-) Schlangen verwalten ihre Elemente nach dem **FIFO**-Prinzip (**F**irst-**I**n-**F**irst-**O**ut).

### Operationen:

<code>boolean isEmpty()</code>	: testet auf Leerheit;
<code>int dequeue()</code>	: liefert erstes Element;
<code>void enqueue(int x)</code>	: reiht $x$ in die Schlange ein;
<code>String toString()</code>	: liefert eine String-Darstellung.

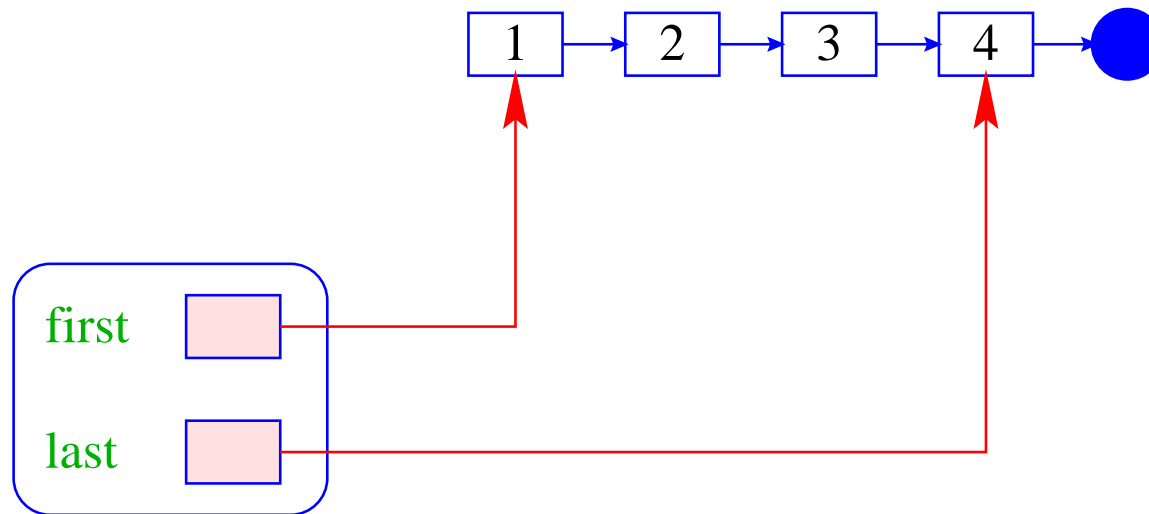
Weiterhin müssen wir eine leere Schlange anlegen können.

Modellierung:

Queue
<ul style="list-style-type: none"><li>+ Queue ()</li><li>+ isEmpty() : boolean</li><li>+ enqueue(x: int) : void</li><li>+ dequeue() : int</li></ul>

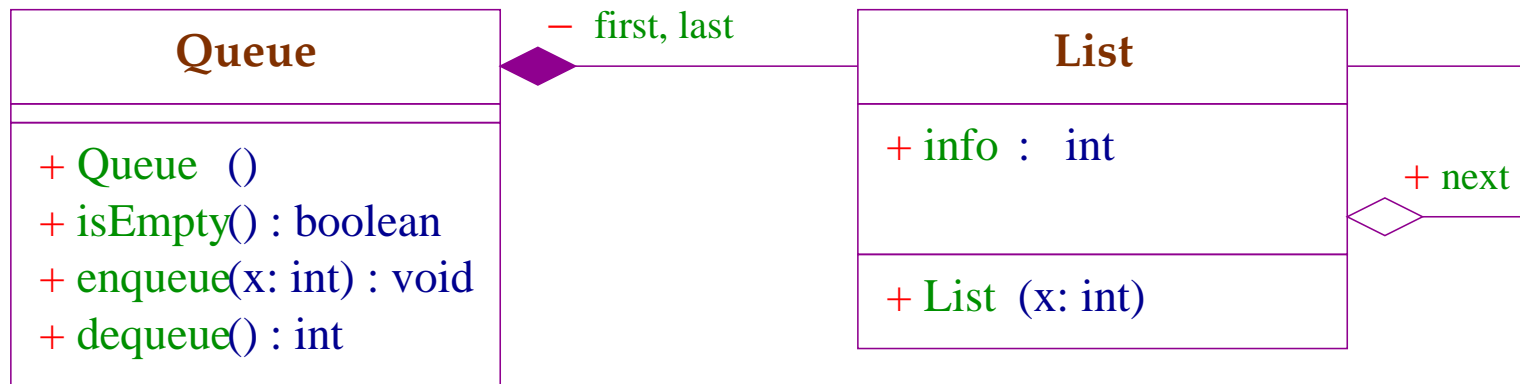
## Erste Idee:

- Realisiere Schlange mithilfe einer Liste :



- `first` zeigt auf das nächste zu entnehmende Element;
- `last` zeigt auf das Element, hinter dem eingefügt wird.

## Modellierung:



Objekte der Klasse **Queue** enthalten **zwei** Verweise auf Objekte der Klasse **List**.

## Implementierung:

```
public class Queue {  
    private List first, last;  
    // Konstruktor:  
    public Queue() {  
        first = last = null;  
    }  
    // Objekt-Methoden:  
    public boolean isEmpty() {  
        return List.isEmpty(first);  
    }  
    ...  
}
```

```

public int dequeue() {
    int result = first.info;
    if (last == first) last = null;
    first = first.next;
    return result;
}

public void enqueue(int x) {
    if (first == null) first = last = new List(x);
    else { last.insert(x); last = last.next; }
}

public String toString() {
    return List.toString(first);
}

} // end of class Queue

```

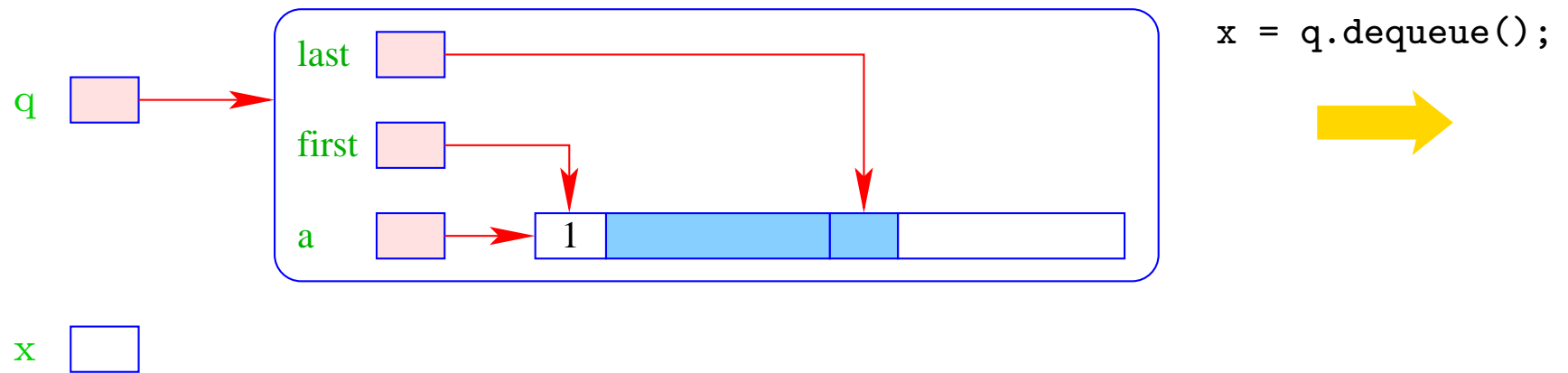
- Die Implementierung ist wieder sehr einfach.
  - ... nutzt ein paar mehr Features von `List` aus;
  - ... die Listen-Elemente sind evt. über den gesamten Speicher verstreut
- ⇒ führt zu schlechtem ↑**Cache**-Verhalten des Programms
- !

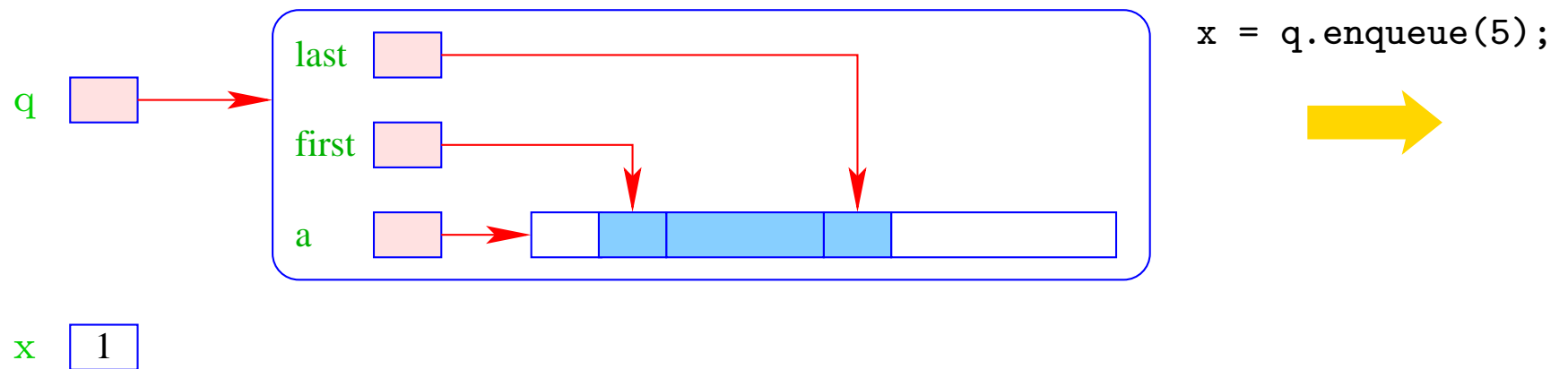


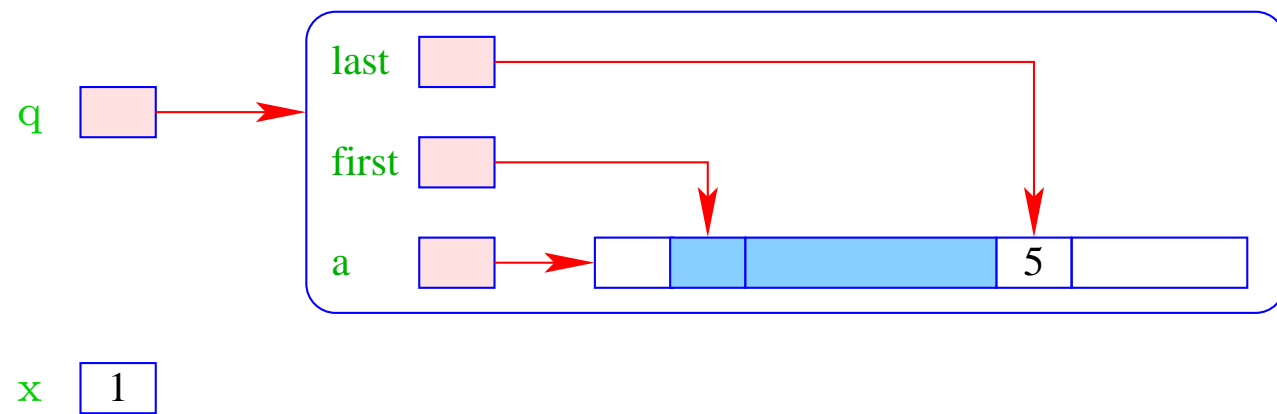
- Die Implementierung ist wieder sehr einfach.
  - ... nutzt ein paar mehr Features von `List` aus;
  - ... die Listen-Elemente sind evt. über den gesamten Speicher verstreut
- ⇒ führt zu schlechtem ↑**Cache**-Verhalten des Programms
- !

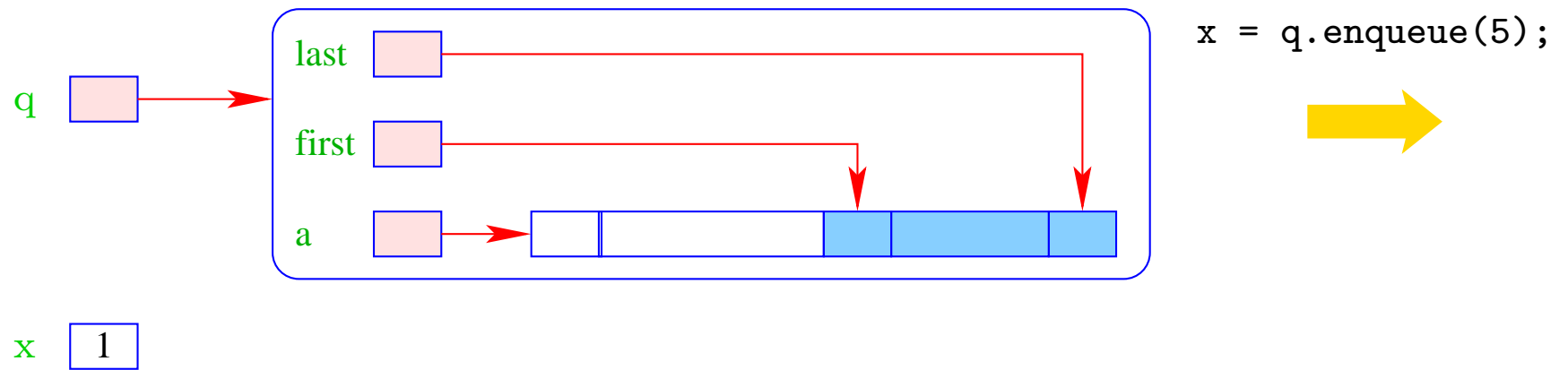
## Zweite Idee:

- Realisiere die Schlange mithilfe eines Felds und **zweier** Pointer, die auf das erste bzw. letzte Element der Schlange zeigen.
- Läuft das Feld über, ersetzen wir es durch ein größeres.



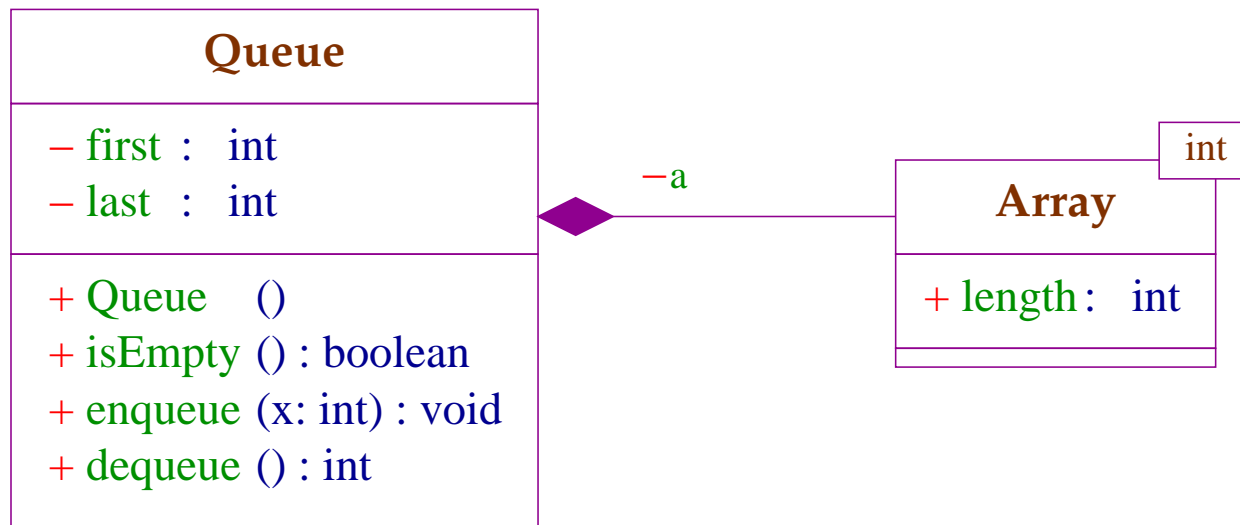








# Modellierung:



## Implementierung:

```
public class Queue {  
    private int first, last;  
    private int[] a;  
    // Konstruktor:  
    public Queue() {  
        first = last = -1;  
        a = new int[4];  
    }  
    // Objekt-Methoden:  
    public boolean isEmpty() { return first==-1; }  
    public String toString() {...}  
    ...  
}
```



## Implementierung von `enqueue()`:

- Falls die Schlange leer war, muss `first` und `last` auf 0 gesetzt werden.
- Andernfalls ist das Feld `a` genau dann voll, wenn das Element `x` an der Stelle `first` eingetragen werden sollte.
- In diesem Fall legen wir ein Feld doppelter Größe an.  
Die Elemente `a[first]`, `a[first+1]`, ..., `a[a.length-1]`,  
`a[0]`, `a[1]`, ..., `a[first-1]` kopieren wir nach `b[0]`, ...,  
`b[a.length-1]`.
- Dann setzen wir `first = 0`;, `last = a.length` und `a = b`;
- Nun kann `x` an der Stelle `a[last]` abgelegt werden.

```

public void enqueue(int x) {
    if (first==-1) {
        first = last = 0;
    } else {
        int n = a.length;
        last = (last+1)%n;
        if (last == first) {
            b = new int[2*n];
            for (int i=0; i<n; ++i) {
                b[i] = a[(first+i)%n];
            } // end for
            first = 0; last = n; a = b;
        } // end if and else
        a[last] = x;
    }
}

```

## Implementierung von `dequeue()` :

- Falls nach Entfernen von `a[first]` die Schlange leer ist, werden `first` und `last` auf `-1` gesetzt.
- Andernfalls wird `first` um 1 (modulo der Länge von `a`) inkrementiert.
- Für eine evt. Freigabe unterscheiden wir zwei Fälle.
- Ist `first < last`, liegen die Schlangen-Elemente an den Stellen `a[first], ..., a[last]`.

Sind dies höchstens  $n/4$ , werden sie an die Stellen `b[0], ..., b[last-first]` kopiert.

```

public int dequeue() {
    int result = a[first];
    if (last == first) {
        first = last = -1;
        return result;
    }

    int n = a.length;
    first = (first+1)%n;
    int diff = last-first;
    if (diff>0 && diff<n/4) {
        int[] b = new int[n/2];
        for(int i=first; i<=last; ++i)
            b[i-first] = a[i];
        last = last-first;
        first = 0; a = b;
    } else ...

```

- Ist `last < first`, liegen die Schlangen-Elemente an den Stellen `a[0], ..., a[last]` und `a[first], ..., a[a.length-1]`.  
Sind dies höchstens  $n/4$ , werden sie an die Stellen `b[0], ..., b[last]` sowie `b[first-n/2], ..., b[n/2-1]` kopiert.
- `first` und `last` müssen die richtigen neuen Werte erhalten.
- Dann kann `a` durch `b` ersetzt werden.

```
if (diff<0 && diff+n<n/4) {  
    int[] b = new int[n/2];  
    for(int i=0; i<=last; ++i)  
        b[i] = a[i];  
    for(int i=first; i<n; i++)  
        b[i-n/2] = a[i];  
    first = first-n/2;  
    a = b;  
}  
return result;  
}
```

## Zusammenfassung:

- Der Datentyp `List` ist nicht sehr **abstract**, dafür extrem flexibel  
     $\implies$  gut geeignet für **rapid prototyping**.
- Für die **nützlichen** (eher) abstrakten Datentypen `Stack` und `Queue` lieferten wir zwei Implementierungen:

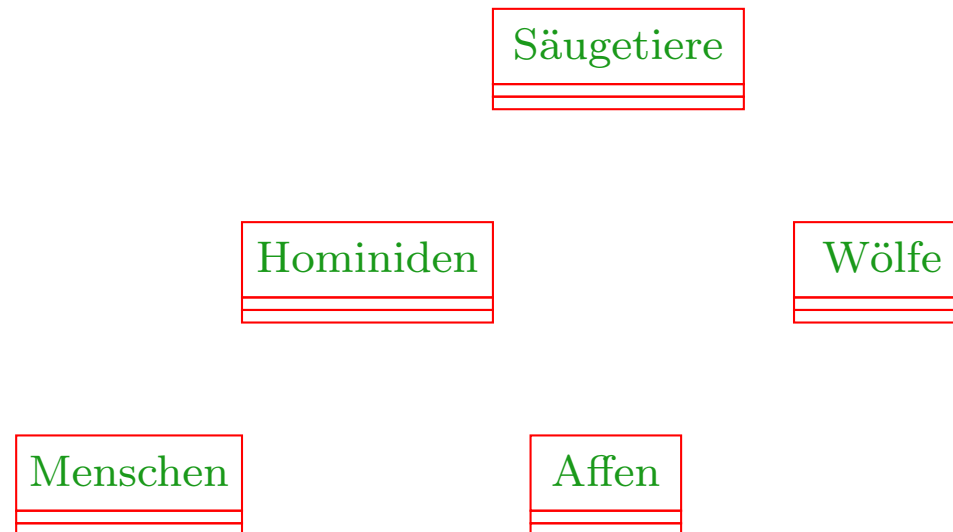
Technik	Vorteil	Nachteil
<code>List</code>	einfach	nicht-lokal
<code>int[]</code>	lokal	etwas komplexer

- **Achtung:** oft werden bei diesen Datentypen noch weitere Operationen zur Verfügung gestellt.

# 11 Vererbung

## Beobachtung:

- Oft werden mehrere Klassen von Objekten benötigt, die zwar ähnlich, aber doch verschieden sind.



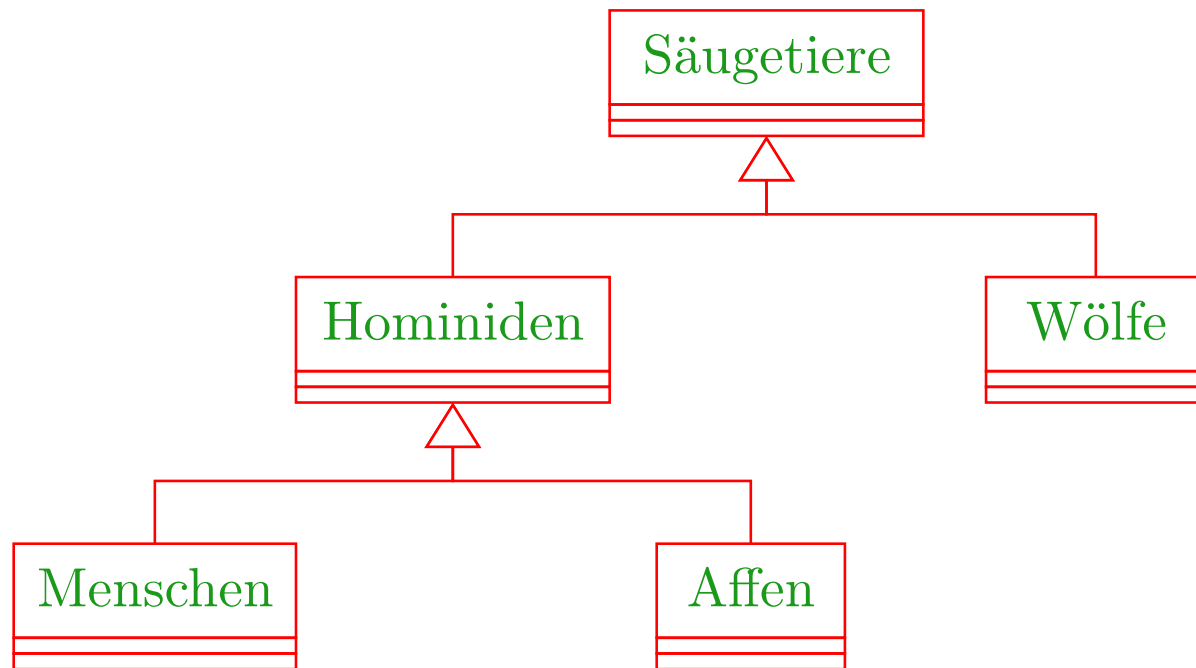


## Idee:

- Finde Gemeinsamkeiten heraus!
- Organisiere in einer Hierarchie!
- Implementiere zuerst was allen gemeinsam ist!
- Implementiere dann nur noch den Unterschied!

⇒ inkrementelles Programmieren

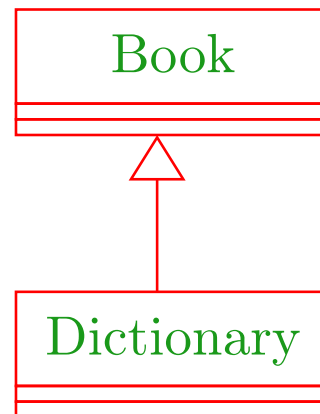
⇒ Software Reuse



## Prinzip:

- Die Unterklasse verfügt über die Members der Oberklasse und eventuell auch noch über weitere.
- Das Übernehmen von Members der Oberklasse in die Unterklasse nennt man **Vererbung** (oder **inheritance**).

## Beispiel:



## Implementierung:

```
public class Book {  
    protected int pages;  
    public Book() {  
        pages = 150;  
    }  
    public void page_message() {  
        System.out.print("Number of pages:\t"+pages+"\n");  
    }  
} // end of class Book  
...
```

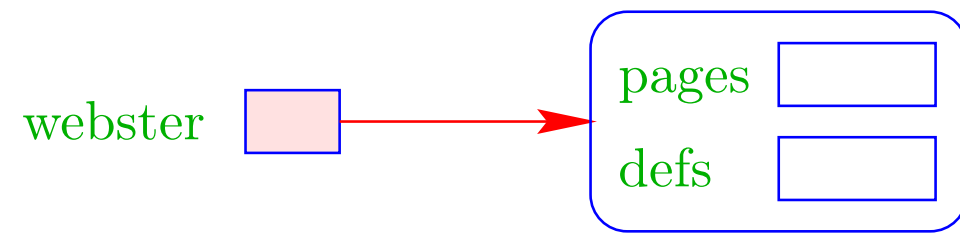
```
public class Dictionary extends Book {  
    private int defs;  
    public Dictionary(int x) {  
        pages = 2*pages;  
        defs = x;  
    }  
    public void defs_message() {  
        System.out.print("Number of defs:\t\t"+defs+"\n");  
        System.out.print("Defs per page:\t\t"+defs/pages+"\n");  
    }  
} // end of class Dictionary
```

- `class A extends B { ... }` deklariert die Klasse `A` als Unterklasse der Klasse `B`.
- Alle Members von `B` stehen damit automatisch auch der Klasse `A` zur Verfügung.
- Als `protected` klassifizierte Members sind auch in der Unterklasse `sichtbar`.
- Als `private` deklarierte Members können dagegen in der Unterklasse `nicht` direkt aufgerufen werden, da sie dort nicht sichtbar sind.
- Wenn ein Konstruktor der Unterklasse `A` aufgerufen wird, wird `implizit` zuerst der Konstruktor `B()` der Oberklasse aufgerufen.

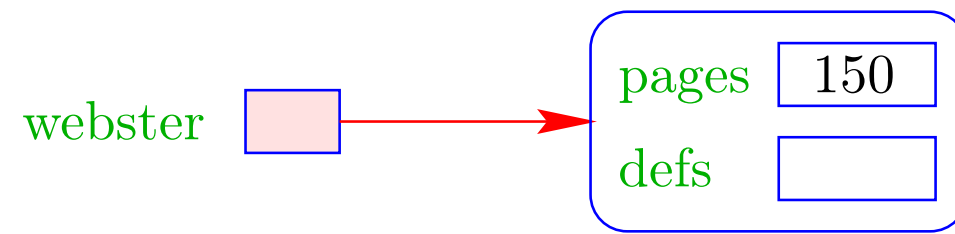
`Dictionary webster = new Dictionary(12400);` liefert:

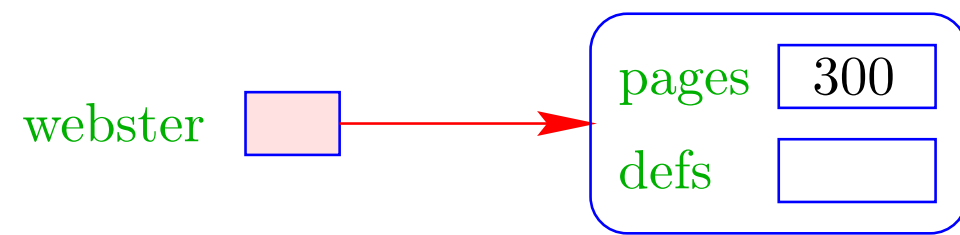
webster

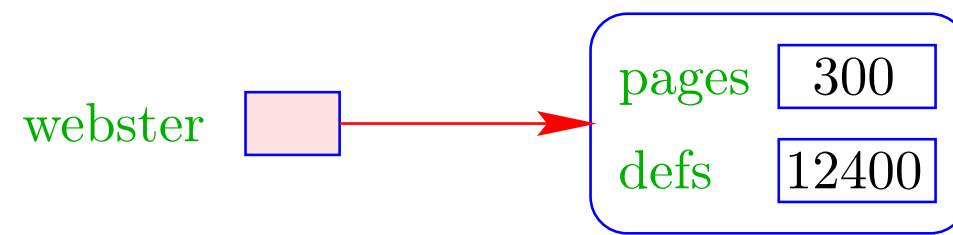












```
public class Words {  
    public static void main(String[] args) {  
        Dictionary webster = new Dictionary(12400);  
        webster.page_message();  
        webster.defs_message();  
    } // end of main  
} // end of class Words
```

- Das neue Objekt `webster` enthält die Attribute `pages` und `defs`, sowie die Objekt-Methoden `page_message()` und `defs_message()`.
- Kommen in der Unterklasse nur weitere Members hinzu, spricht man von einer `is_a`-Beziehung. (Oft müssen aber Objekt-Methoden der Oberklasse in der Unterklasse `umdefiniert` werden.)

- Die Programm-Ausführung liefert:

Number of pages:	300
Number of defs:	12400
Defs per page:	41

## 11.1 Das Schlüsselwort `super`

- Manchmal ist es erforderlich, in der Unterklasse **explizit** die Konstruktoren oder Objekt-Methoden der Oberklasse aufzurufen. Das ist der Fall, wenn
  - Konstruktoren der Oberklasse aufgerufen werden sollen, die Parameter besitzen;
  - Objekt-Methoden oder Attribute der Oberklasse und Unterklasse gleiche Namen haben.
- Zur Unterscheidung der aktuellen Klasse von der Oberklasse dient das Schlüsselwort `super`.

... im Beispiel:

```
public class Book {  
    protected int pages;  
    public Book(int x) {  
        pages = x;  
    }  
    public void message() {  
        System.out.print("Number of pages:\t"+pages+"\n");  
    }  
} // end of class Book  
...
```

```
public class Dictionary extends Book {
    private int defs;
    public Dictionary(int p, int d) {
        super(p);
        defs = d;
    }
    public void message() {
        super.message();
        System.out.print("Number of defs:\t\t"+defs+"\n");
        System.out.print("Defs per page:\t\t"+defs/pages+"\n");
    }
} // end of class Dictionary
```



- `super(...);` ruft den entsprechenden Konstruktor der Oberklasse auf.
- Analog gestattet `this(...);` den entsprechenden Konstruktor der eigenen Klasse aufzurufen.
- Ein solcher expliziter Aufruf muss stets ganz am Anfang eines Konstruktors stehen.
- Deklariert eine Klasse `A` einen Member `memb` gleichen Namens wie in einer Oberklasse, so ist nur noch der Member `memb` aus `A` sichtbar.
- Methoden mit unterschiedlichen Argument-Typen werden als verschieden angesehen.
- `super.memb` greift für das aktuelle Objekt `this` auf Attribute oder Objekt-Methoden `memb` der Oberklasse zu.
- Eine andere Verwendung von `super` ist nicht gestattet.

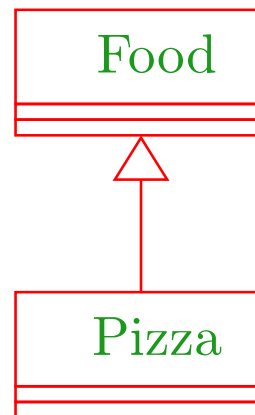
```
public class Words {  
    public static void main(String[] args) {  
        Dictionary webster = new Dictionary(540,36600);  
        webster.message();  
    } // end of main  
} // end of class Words
```

- Das neue Objekt `webster` enthält die Attribute `pages` wie `defs`.
- Der Aufruf `webster.message()` ruft die Objekt-Methode der Klasse `Dictionary` auf.
- Die Programm-Ausführung liefert:

Number of pages:	540
Number of defs:	36600
Defs per page:	67

## 11.2 Private Variablen und Methoden

Beispiel:



Das Programm `Eating` soll die Anzahl der Kalorien pro Mahlzeit ausgeben.

```
public class Eating {  
    public static void main (String[] args) {  
        Pizza special = new Pizza(275);  
        System.out.print("Calories per serving: " +  
            special.calories_per_serving());  
    } // end of main  
} // end of class Eating
```

```
public class Food {  
    private int CALORIES_PER_GRAM = 9;  
    private int fat, servings;  
    public Food (int num_fat_grams, int num_servings) {  
        fat = num_fat_grams;  
        servings = num_servings;  
    }  
    private int calories() {  
        return fat * CALORIES_PER_GRAM;  
    }  
    public int calories_per_serving() {  
        return (calories() / servings);  
    }  
} // end of class Food
```

```
public class Pizza extends Food {  
    public Pizza (int amount_fat) {  
        super (amount_fat,8);  
    }  
} // end of class Pizza
```

- Die Unterklasse `Pizza` verfügt über alle Members der Oberklasse `Food` – wenn auch nicht alle **direkt** zugänglich sind.
- Die Attribute und die Objekt-Methode `calories()` der Klasse `Food` sind privat, und damit für Objekte der Klasse `Pizza` verborgen.
- Trotzdem können sie von der `public` Objekt-Methode `calories_per_serving` benutzt werden.

```
public class Pizza extends Food {  
    public Pizza (int amount_fat) {  
        super (amount_fat,8);  
    }  
} // end of class Pizza
```

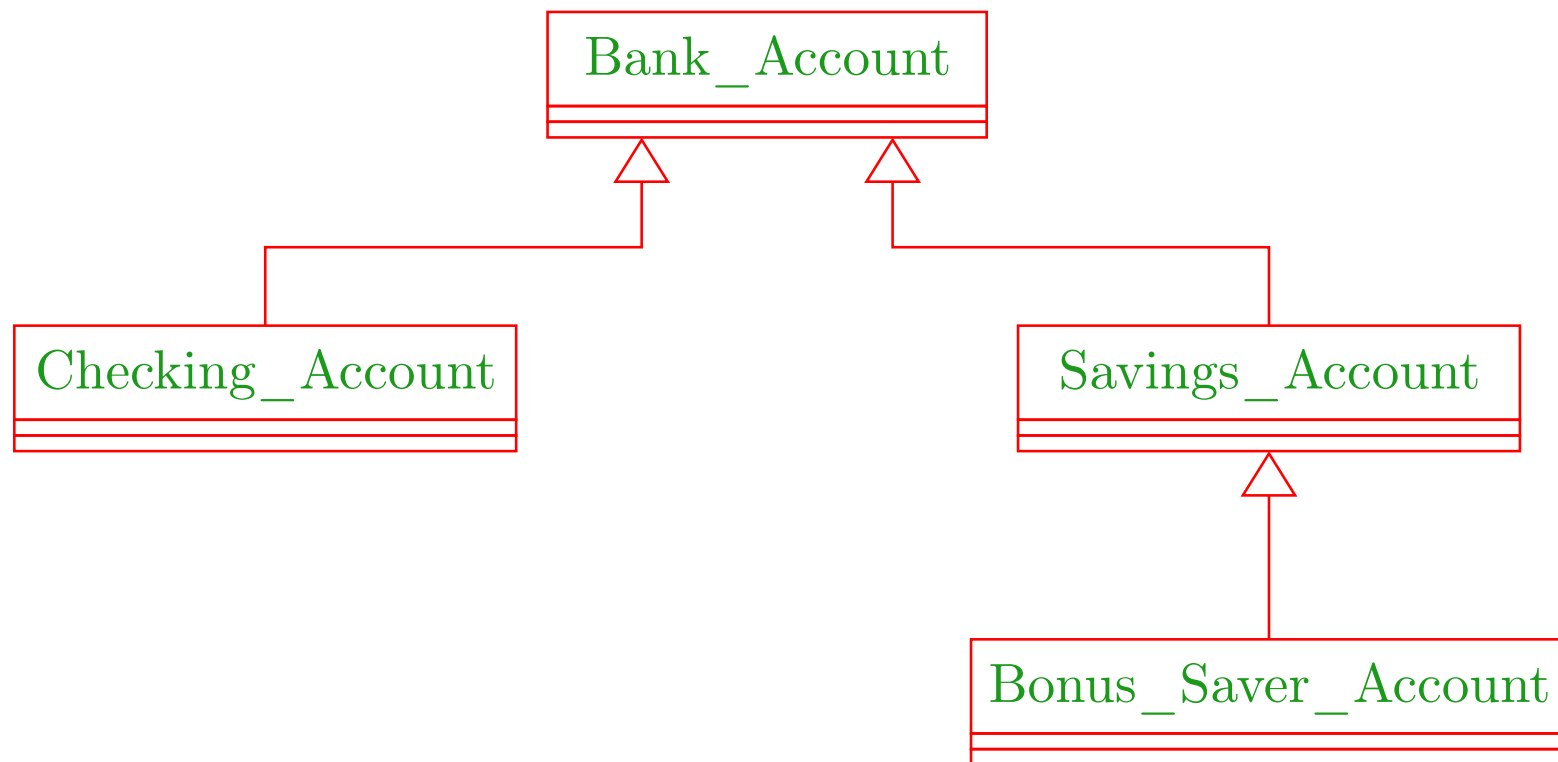
- Die Unterklasse `Pizza` verfügt über alle Members der Oberklasse `Food` – wenn auch nicht alle **direkt** zugänglich sind.
- Die Attribute und die Objekt-Methode `calories()` der Klasse `Food` sind privat, und damit für Objekte der Klasse `Pizza` verborgen.
- Trotzdem können sie von der `public` Objekt-Methode `calories_per_serving` benutzt werden.

... Ausgabe des Programms:

Calories per serving: 309

## 11.3 Überschreiben von Methoden

Beispiel:





## Aufgabe:

- Implementierung von einander abgeleiteter Formen von Bank-Konten.
- Jedes Konto kann eingerichtet werden, erlaubt Einzahlungen und Auszahlungen.
- Verschiedene Konten verhalten sich unterschiedlich in Bezug auf Zinsen und Kosten von Konto-Bewegungen.

## Einige Konten:

```
public class Bank {  
    public static void main(String[] args) {  
        Savings_Account savings =  
            new Savings_Account (4321, 5028.45, 0.02);  
        Bonus_Saver_Account big_savings =  
            new Bonus_Saver_Account (6543, 1475.85, 0.02);  
        Checking_Account checking =  
            new Checking_Account (9876, 269.93, savings);  
        ...  
    }  
}
```

## Einige Konto-Bewegungen:

```
savings.deposit (148.04);  
big_savings.deposit (41.52);  
savings.withdraw (725.55);  
big_savings.withdraw (120.38);  
checking.withdraw (320.18);  
    } // end of main  
} // end of class Bank
```

## Einige Konto-Bewegungen:

```
savings.deposit (148.04);  
big_savings.deposit (41.52);  
savings.withdraw (725.55);  
big_savings.withdraw (120.38);  
checking.withdraw (320.18);  
    } // end of main  
} // end of class Bank
```

Fehlt nur noch die Implementierung der Konten selbst.

```

public class Bank_Account {
    // Attribute aller Konten-Klassen:
    protected int account;
    protected double balance;
    // Konstruktor:
    public Bank_Account (int id, double initial) {
        account = id; balance = initial;
    }
    // Objekt-Methoden:
    public void deposit(double amount) {
        balance = balance+amount;
        System.out.print("Deposit into account "+account+"\n"
            +"Amount:\t\t"+amount+"\n"
            +"New balance:\t"+balance+"\n\n");
    }
    ...
}

```

- Anlegen eines Kontos `Bank_Account` speichert eine (hoffentlich neue) Konto-Nummer sowie eine Anfangs-Einlage.
- Die zugehörigen Attribute sind `protected`, d.h. können nur von Objekt-Methoden der Klasse bzw. ihrer Unterklassen modifiziert werden.
- die Objekt-Methode `deposit` legt Geld aufs Konto, d.h. modifiziert den Wert von `balance` und teilt die Konto-Bewegung mit.

```
public boolean withdraw(double amount) {  
    System.out.print("Withdrawal from account "+ account +"\n"  
        +"Amount:\t\t"+ amount +"\n");  
    if (amount > balance) {  
        System.out.print("Sorry, insufficient funds...\n\n");  
        return false;  
    }  
    balance = balance-amount;  
    System.out.print("New balance:\t"+ balance +"\n\n");  
    return true;  
}  
} // end of class Bank_Account
```

- Die Objekt-Methode `withdraw()` nimmt eine Auszahlung vor.
- Falls die Auszahlung scheitert, wird eine Mitteilung gemacht.
- Ob die Auszahlung erfolgreich war, teilt der Rückgabewert mit.
- Ein `Checking_Account` verbessert ein normales Konto, indem im Zweifelsfall auf die Rücklage eines Sparkontos zurückgegriffen wird.



## Ein Giro-Konto:

```
public class Checking_Account extends Bank_Account {
    private Savings_Account overdraft;
// Konstruktor:
    public Checking_Account(int id, double initial,
                           Savings_Account savings) {
        super (id, initial);
        overdraft = savings;
    }
    ...
}
```

```
// modifiziertes withdraw():
public boolean withdraw(double amount) {
    if (!super.withdraw(amount)) {
        System.out.print("Using overdraft...\n");
        if (!overdraft.withdraw(amount-balance)) {
            System.out.print("Overdraft source insufficient.\n\n");
            return false;
        } else {
            balance = 0;
            System.out.print("New balance on account "+ account +": 0\n\n");
        }
    }
    return true;
}
} // end of class Checking_Account
```

- Die Objekt-Methode `withdraw` wird neu definiert, die Objekt-Methode `deposit` wird übernommen.
- Der Normalfall des Abhebens erfolgt (als Seiteneffekt) beim Testen der ersten `if`-Bedingung.
- Dazu wird die `withdraw`-Methode der Oberklasse aufgerufen.
- Scheitert das Abheben mangels Geldes, wird der Fehlbetrag vom Rücklagen-Konto abgehoben.
- Scheitert auch das, erfolgt keine Konto-Bewegung, dafür eine Fehlermeldung.
- Andernfalls sinkt der aktuelle Kontostand auf 0 und die Rücklage wird verringert.

## Ein Sparbuch:

```
public class Savings_Account extends Bank_Account {
    protected double interest_rate;
    // Konstruktor:
    public Savings_Account (int id, double init, double rate) {
        super(id,init); interest_rate = rate;
    }
    // zusaetzliche Objekt-Methode:
    public void add_interest() {
        balance = balance * (1+interest_rate);
        System.out.print("Interest added to account: "+ account
            +"\nNew balance:\t"+ balance +"\n\n");
    }
} // end of class Savings_Account
```

- Die Klasse `Savings_Account` erweitert die Klasse `Bank_Account` um das zusätzliche Attribut `double interest_rate` (Zinssatz) und eine Objekt-Methode, die die Zinsen gutschreibt.
- Alle sonstigen Attribute und Objekt-Methoden werden von der Oberklasse geerbt.
- Die Klasse `Bonus_Saver_Account` erhöht zusätzlich den Zinssatz, führt aber Strafkosten fürs Abheben ein.

## Ein Bonus-Sparbuch:

```
public class Bonus_Saver_Account extends Savings_Account {
    private int penalty;
    private double bonus;
    // Konstruktor:
    public Bonus_Saver_Account(int id, double init, double rate) {
        super(id, init, rate); penalty = 25; bonus = 0.03;
    }
    // Modifizierung der Objekt-Methoden:
    public boolean withdraw(double amount) {
        System.out.print("Penalty incurred:\t"+ penalty +"\n");
        return super.withdraw(amount+penalty);
    }
    ...
}
```

```
public void add_interest() {  
    balance = balance * (1+interest_rate+bonus);  
    System.out.print("Interest added to account: "+ account  
                    +"\nNew balance:\t" + balance +"\n\n");  
}  
} // end of class Bonus_Safer_Account
```

... als [Ausgabe](#) erhalten wir dann:

Deposit into account 4321

Amount: 148.04

New balance: 5176.49

Deposit into account 6543

Amount: 41.52

New balance: 1517.37

Withdrawal from account 4321

Amount: 725.55

New balance: 4450.94



Penalty incurred: 25  
Withdrawal from account 6543  
Amount: 145.38  
New balance: 1371.98999999999998

Withdrawal from account 9876  
Amount: 320.18  
Sorry, insufficient funds...

Using overdraft...  
Withdrawal from account 4321  
Amount: 50.25  
New balance: 4400.69

New balance on account 9876: 0