Behälter

Behälter, Mengen, Generische Klassen, Iteratoren, foreach, mitwachsende Mengen, Maps

Mögliche Merkmale von Behälter

- keine Reihenfolge:
  - Mengen
  - Bags
- mit Standardreihenfolge:
  - Bäume
  - Listen
- mit direktem Zugriff:
  - Maps
  - Arrays
- mit Einfüge/Entnahmereihenfolge:
  - Stacks
  - Queues

Behälter ohne Standard-Reihenfolge

- Menge
  - jedes Element ist höchstens einmal vorhanden
- Bag
  - Elemente können mehrmals ($n \geq 0$) vorhanden sein
- fuzzy Set
    - Elemente mit Gewissheit r ($0 \leq r \leq 1$) vorhanden

Vorteile und Nachteile:

- ✓ einfügen und entfernen von Elementen ist leicht,
- ✓ Reihenfolge des Ein/Ausfügens ist beliebig
- \- suchen eines Elements ist schwierig

Behälter mit Direktzugriff

- Array
  - schneller Direktzugriff
  - unveränderliche Größe
  - Index immer Intervall [0 .. length -1]

- ArrayList (Klasse in java.util)
  - Liste mit Direktzugriff
  - veränderliche Größe

- Map – oder *assoziativer Array*
  - veränderliche Größe
  - Index eine beliebige Aufzählung
  - In Java als Interface
    - plus einige Implementierungen
  - Sprachbestandteil z. B. von php :

Java:

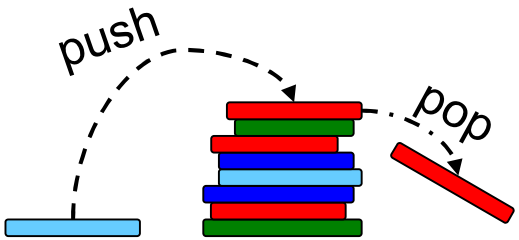
```
for (int i = 0; i < 5; i++)  list1.add(0,  „Nr.: " + i);
// Show list
for (int i = 0; i < list1.size(); i++)
        System.out.println(list1.get(i));
```

PHP:

```
$hauptstadt = array( "NY" => "Albany",
                     "CA" => "San Francisco",
                     "TX" => "Austin" );

print „Hauptstadt von Texas ist " . $hauptstadt["TX"];
```

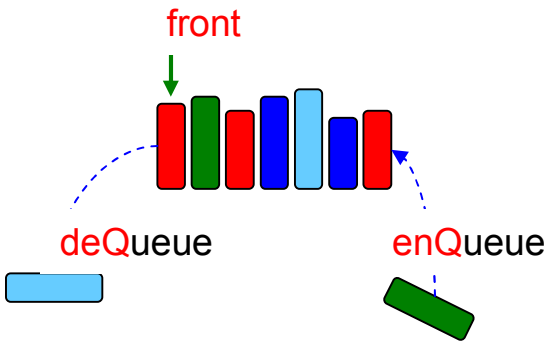
Entnahme/Einfügereihenfolge

- Stacks
  - last-in-first-out
  - Nützlich für Evaluierung von Ausdrücken



- Queues
  - first-in-first-out
  - Warteschlangen, Puffer

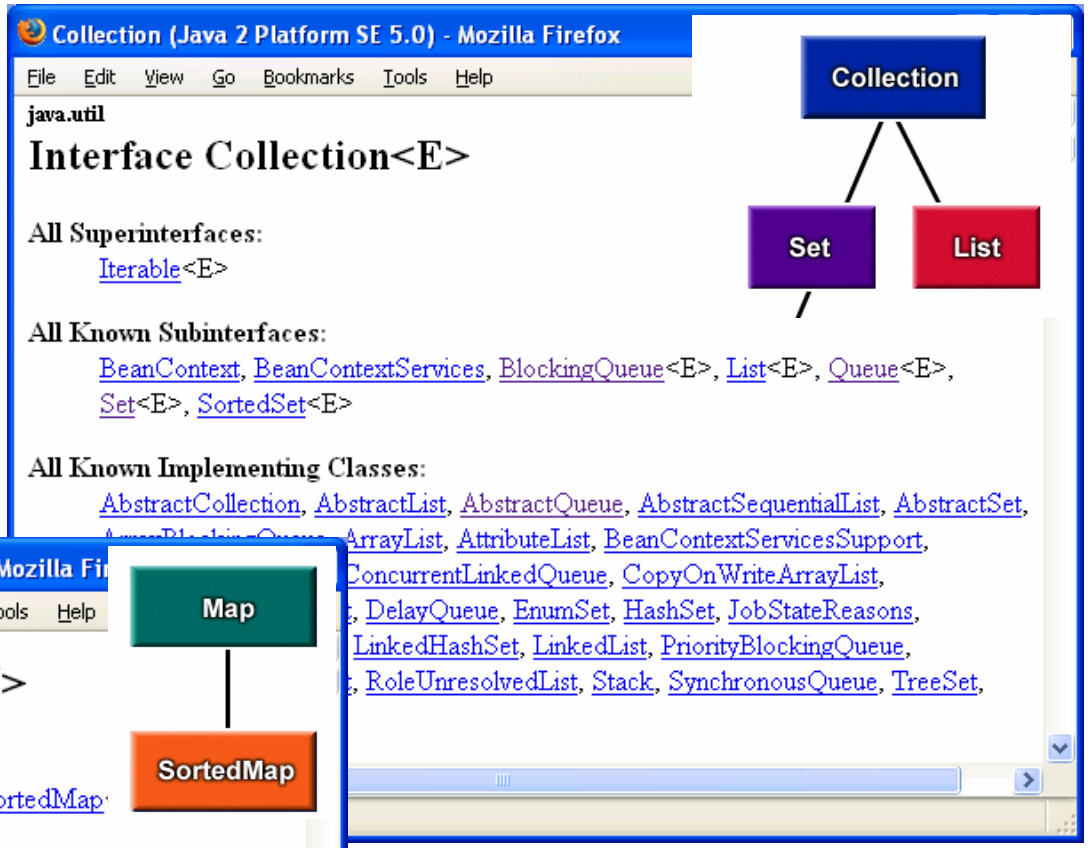
Varianten:

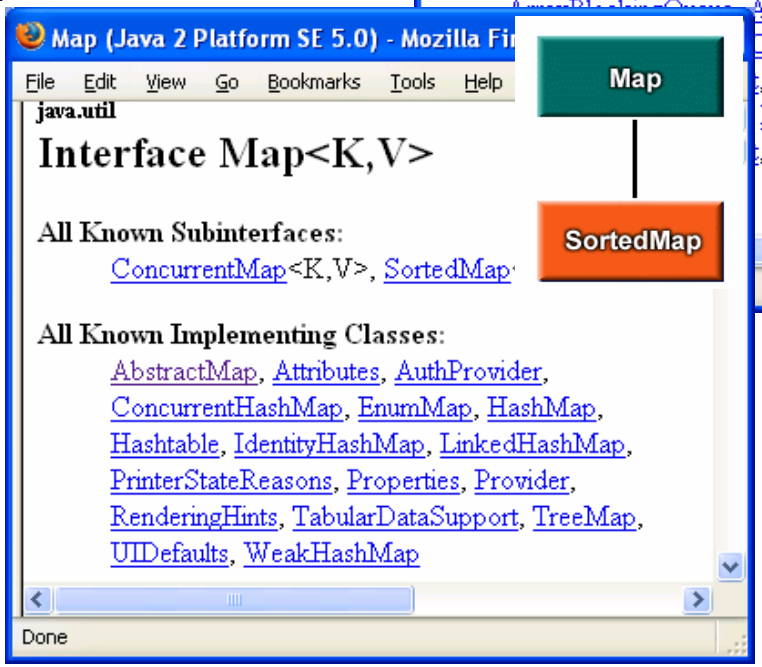


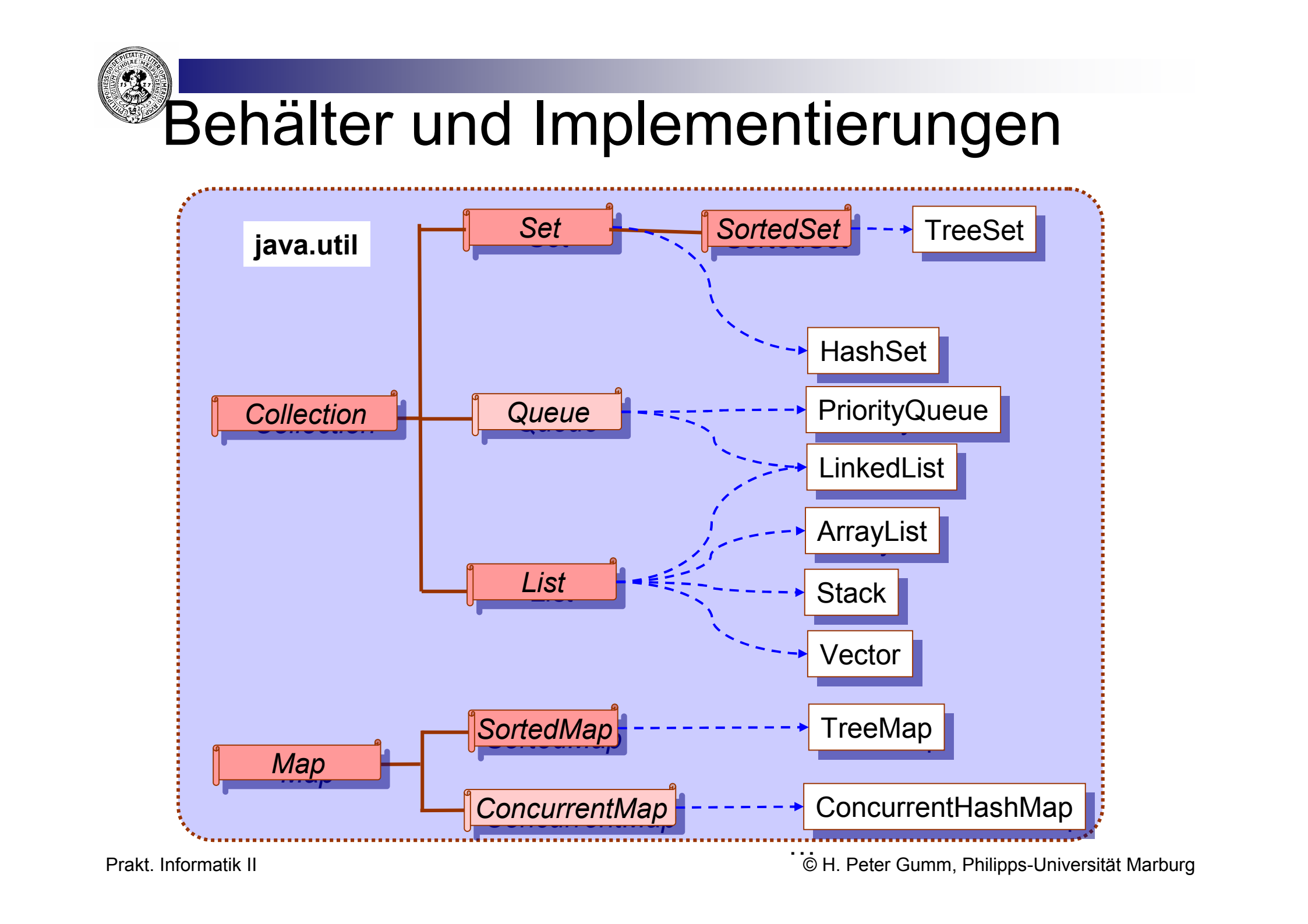
- Deque (double ended queue)
  - Entfernen und einfügen vorn und hinten
- aktive Puffer
  - *Blockieren* bei Entnahmeversuch leer
  - Entfernender Prozess *wacht auf*, wenn Objekt kommt.
- Priority Queue
  - ein Heap – nach Eingangsdatum geordnet

Kern des Java-Collection Interface

- Zwei Hierarchien von interfaces
- dazu implementierende Klassen
- map nicht als spezielle Menge implementiert
  - Geschmackssache
- Alle Behälter besitzen Methoden um
  - die Elemente zu durchlaufen
  - Element zu löschen
  - einzufügen



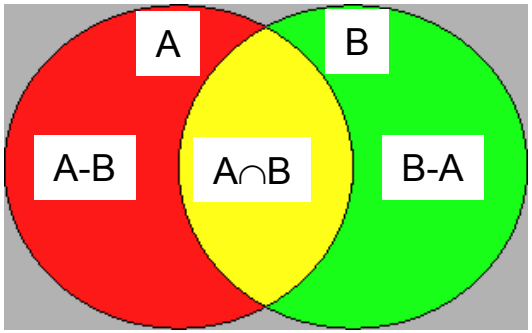




Behälter und Implementierungen

Beispiel: Mengen - selbstgemacht

- Mengen sind „Zusammenfassungen gleichartiger Elemente“
- Endliche Mengen gegeben durch Aufzählung
  - $\{x_1,x_2,x_3,x_4\}$, $\{0,1,2,3,4\}$, $\{42\}$, $\{\}$ = $\varnothing$
- Operationen
  - Vereinigung, Schnitt, Differenz
    - $S \cup T$, $S \cap T$, $S-T$
  - Element x einfügen
    - $S \cup \{x\}$
- Selektoren, Observatoren
  - $x \in S$: „S contains x“
  - $S \subseteq T$: „S ist Teilmenge (subset) von T“



Entscheidungen

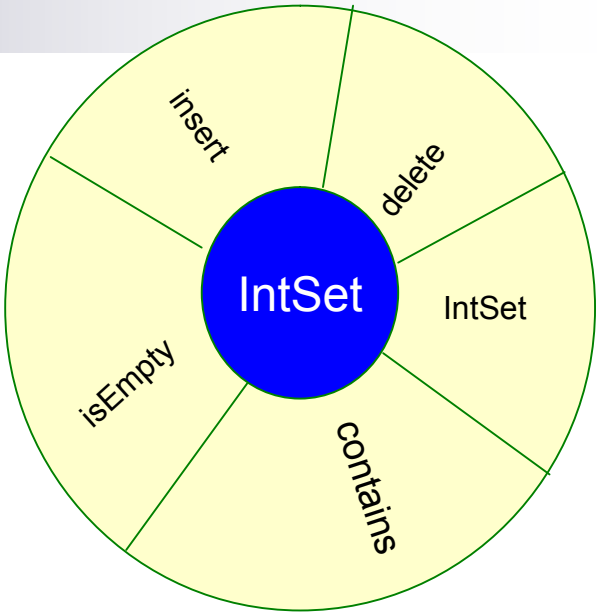
- Klasse IntSet
- Methoden als Objektmethoden
  - Beispiel:
    - insert: $IntSet \times int \rightarrow IntSet$
- Entscheidung:
  - Implementierung als *Werttyp*:
    - insert liefert neue Menge
      - `IntSet insert(int i)`
  - oder Implementierung als *veränderlicher Typ*:
    - insert modifiziert Ausgangsmenge S
      - `void insert(int i)`
- Wir begrenzen die Größe der Menge
  - Ist die Menge voll, so liefert insert eine Exception

Datentyp *IntSet*

- Menge von Integern
- Klasse: IntSet

- Operationen
  - insert: $IntSet \times int \rightarrow IntSet$
  - delete: $IntSet \times int \rightarrow IntSet$
  - theEmptySet : $\rightarrow IntSet$
  - contains: $IntSet \times int \rightarrow boolean$
  - isEmpty: $IntSet \rightarrow boolean$

- Gemeint sind die folgenden Operationen:
  - theEmptySet = $\varnothing$
  - insert(S,i) = S $\cup$ { i }
  - delete(S,i) = S \ {i}
  - contains(S,i) = i $\in$ S
  - isEmpty = (S $\equiv \varnothing$)



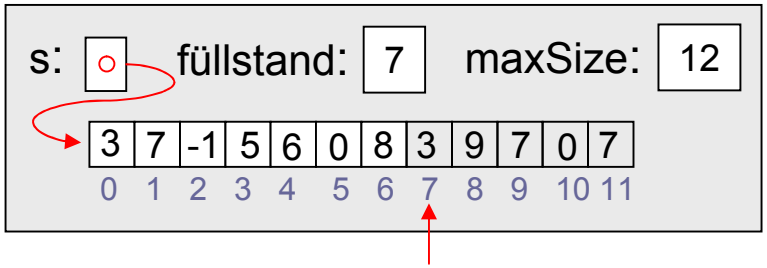
IntSet

«Konstruktoren»
+ IntSet( )
«Mutatoren»
+ void insert(int)
+ void delete(int)
«Selektoren»
+ boolean contains(int)
+ boolean isEmpty(int)
....

mutable Type

IntSet als *veränderlicher Typ*

die Menge { 3, 7,-1, 5, 7, 0, 8 }



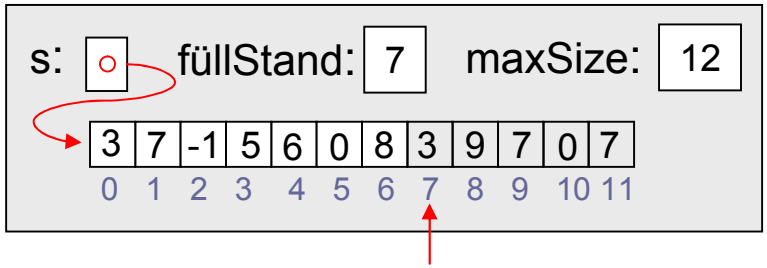
- füllStand zeigt auf nächste freie Position
- repräsentiert Menge: { s[i] | 0 ≤ i < füllStand}
- Elemente s[j] mit j ≥ füllStand nicht in Menge

- Neue Java-Syntax:
  - varargs: Methodensignatur für Aufruf mit variabler Anzahl von Parametern
    - *Resultattyp* **(***typ* **...** *elemente***)**
    - *elemente* ist damit ein array **typ[]**
  - foreach-Schleife über einen Array
    - **for(***typ elt* **:** *behälter***)** *block*

```
7  public class IntSet{
8  // Klassenfelder
9     static int maxSize=100;
10 // Objektfelder
11    /** Die Menge wird repräsentiert
12     *  durch  s[0 .. füllStand-1] */
13    private int[] s;
14    private int füllStand;
15
16 // Klasseninvariante
17    boolean nichtsDoppelt(){
18       for(int i=0;i<füllStand-1;i++)
19         for(int j=i+1;j<füllStand;j++)
20           if(s[i]==s[j]) return false;
21       return true;
22    }
23
24   /** Konstruiere die leere Menge */
25    public IntSet(){
26       s = new int[maxSize];
27       füllStand=0;
28    }
29   /** Konstruktor für nichtleere Menge;*/
30    public IntSet(int ... zahlen)
31                          throws SetException{
32      this();
33      for(int e:zahlen){ insert(e); }
34      // Prüfe Integrität
35      assert nichtsDoppelt();
36    }
```

IntSet als *veränderlicher Typ*

die Menge { 3, 7,-1, 5, 7, 0, 8 }

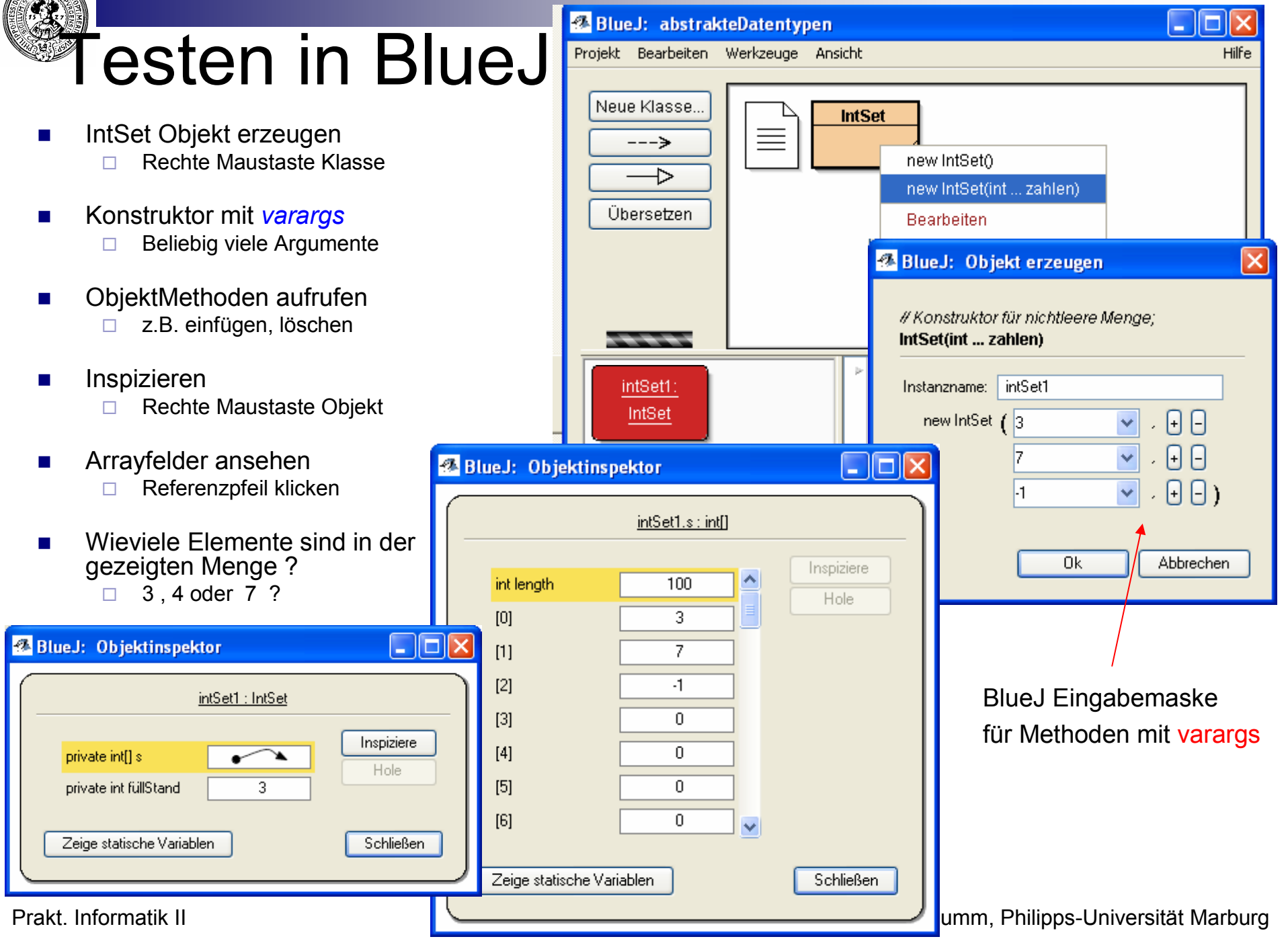


- Abschnitt s[0 .. füllStand-1] repräsentiert Menge
- Assertions prüfen bei Einfügen, dass das Element noch nicht vorhanden ist
- Falls kein Platz mehr, wird Exception geworfen.
- Löschen: Element wird durch s[füllStand-1] überschrieben und füllStand dekrementiert.
- assertion – zur Sicherheit z.B. wenn der Programmiererkollege einmal *search* ändern will ….

```
/** Gib Index i mit s[i]==z --- oder -1.*/
private int search(int z){
    for(int i=0; i<füllStand; i++)
        if(s[i] == z) return i;
    return -1;
}

/** Prüfe, ob Element z in der Menge */
public boolean contains(int z){
    return search(z) >= 0;
}

/** Füge Element z ein, falls nicht vorhanden */
public void insert(int z) throws SetException{
    if(füllStand>=maxSize)
        throw new SetException("Menge voll");
    else if(!contains(z)) s[füllStand++]=z;
    // Pruefe Integrität
    assert nichtsDoppelt();
}

/** Lösche z, falls vorhanden */
public void delete(int z){
  int idx = search(z);
  if( idx >= 0){ assert idx < füllStand;
      füllStand--;
      s[idx] = s[füllStand];
   }
}
```

Testen in BlueJ

- IntSet Objekt erzeugen
  - Rechte Maustaste Klasse
- Konstruktor mit *varargs*
  - Beliebig viele Argumente
- ObjektMethoden aufrufen
  - z.B. einfügen, löschen
- Inspizieren
  - Rechte Maustaste Objekt
- Arrayfelder ansehen
  - Referenzpfeil klicken
- Wieviele Elemente sind in der gezeigten Menge ?
  - 3 , 4 oder 7 ?



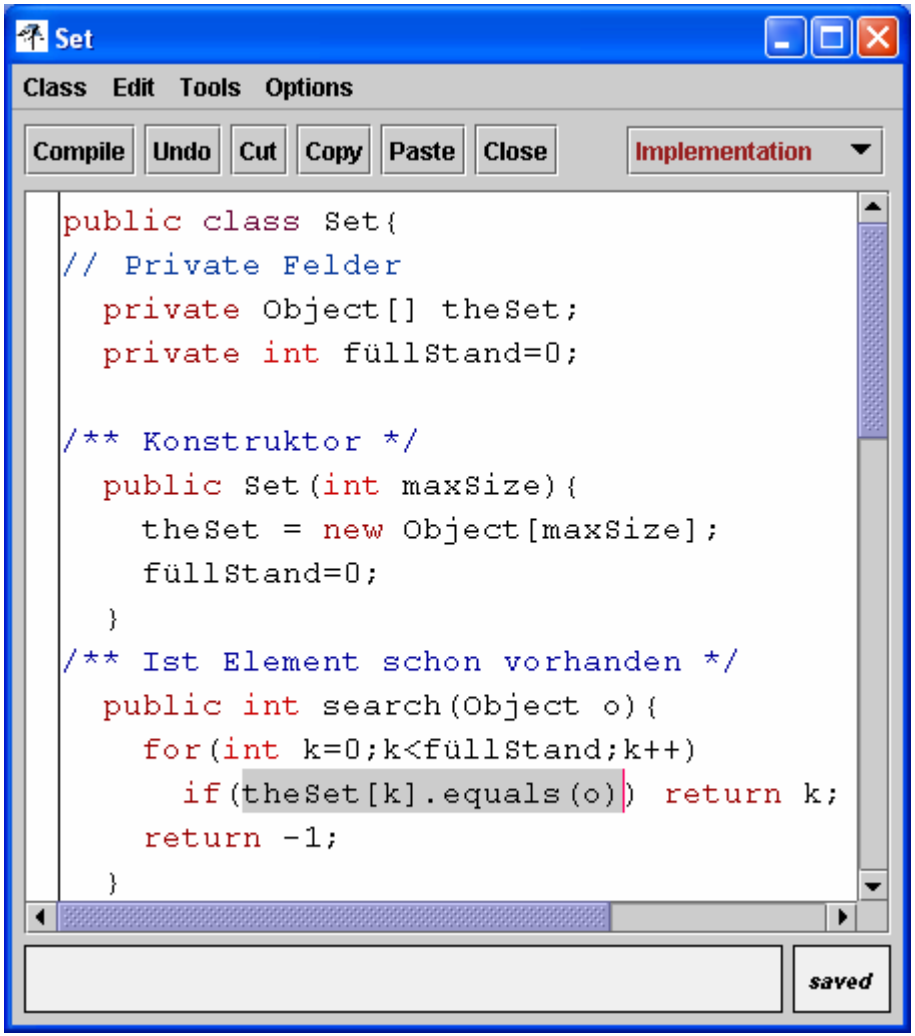
BlueJ Eingabemaske
für Methoden mit varargs

Effizienz der Operationen

- N: Größe der Menge S
  - insert : O(N)
  - contains : O(N)
  - delete : O(N)
- Wenn man die Elemente sortiert hält:
  - insert: O(N)
    - Position suchen O(N)
    - Platz schaffen O(N)
    - Element kopieren O(1)
  - contains: O(log(n))
    - binSearch O(log(N))
  - delete O(N)
    - Element suchen O(log(n))
    - Elemente zusammenschieben O(N)
- Kann man auch erreichen:
  - Insert O(1) ?
  - Was ist dann mit
    - contains,
    - delete

Mengen von Objekten

- Man kann Mengen beliebiger Objekte genauso implementieren
  - private Object[ ] s;
- Für Vergleiche ersetze „==„ durch „equals(…)“
- Nachteil:
  - Elemente haben Typ verloren
  - Alle sind vom Typ *Object.*
  - Entnommene Elemente müssen vor Weiterverarbeitung „gecastet“ werden

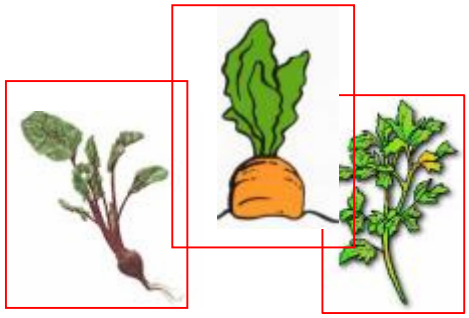


Mitwachsende Mengen

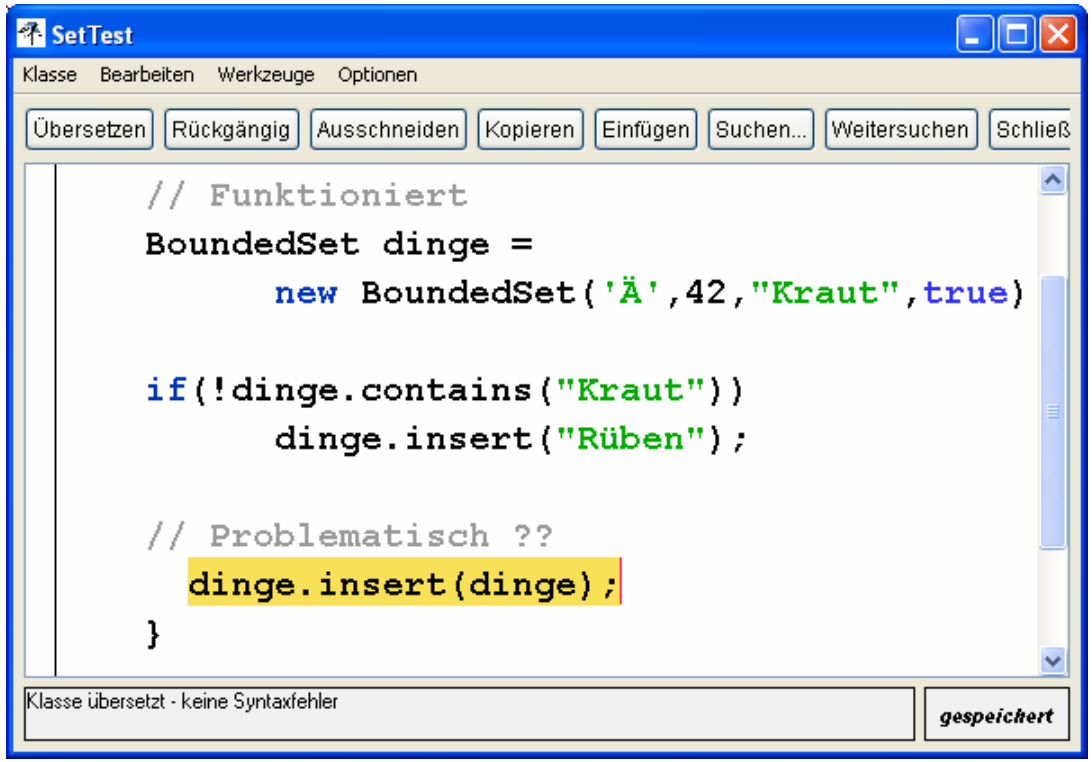
- Unbegrenzt große Mengen erhalten wir durch den folgenden Trick.
  - Er funktioniert in Java, weil Arrays dynamisch angelegt werden.
  - In vielen anderen Sprachen muss die Größe eines Arrays schon zur Compile-Zeit feststehen.
- Wenn beim Einfügen die Menge überlaufen würde, wird sie dynamisch vergrößert
- Analog könnte man beim Entfernen eines Elementes die Größe des Arrays halbieren, falls die Menge nur noch ¼ voll ist.

```
}
/** Element einfügen */
public void insert(Object o){
  if(!this.contains(o)){
    if (füllStand==theSet.length){
     // vergrößere zuerst den Array
     Object[] temp = new Object[2*theSet.length];
     System.arraycopy(theSet,0,temp,0,theSet.length);
     theSet=temp;
     this.insert(o);
    }else {
       theSet[füllStand]=o;
      füllStand++;
    }
  }
```

Kraut und Rüben



- Primitive Typen werden automatisch in Objekte umgewandelt
  - autoboxing / autounboxing macht es möglich
- BoundedSets können beliebige Objekte aufnehmen – es lässt sich nicht verbieten
  - Strings, Zahlen, booleans, …
  - Punkte, Konten, BoundedSets, …
  - sogar sich selber !!!
- Nachteil:
  - Elemente haben ihren Typ verloren
  - Alle sind vom Typ *Object.*
  - Müssen notfalls wieder „gecastet“ werden
- Besser wäre Ordnung und Kontrolle
  - Mengen von *Integer*
  - Mengen von *Punkten*
  - Mengen von ElementTyp ***E***
  - aber bitte nicht für jeden neuen Typ *E* eine neue Implementierung



Generische Klassen - Deklaration

- Feature ab Java1.5 (bzw. 5.0)
  - noch nicht ganz ausgereift
- Deklaration
  - class BoundedSet<X>{ … }
  - class Pair<X,Y>{ … }
- Parameter steht für festen Typ. Erlaubt: alles was *für jeden Objekttyp* geht, u.a.:
  - X element;
  - for(X e : s){ … }
  - X x = element ;
  - x.equals(element)
  - x.toString()
- **Nicht erlaubt**:
  - X e = new X();
  - X[ ] xs = new X[maxSize];
- Abhilfe: Casting
  - X e = (X)(new Object());
  - X[ ] s = (X[ ])(new Object[maxSize]);

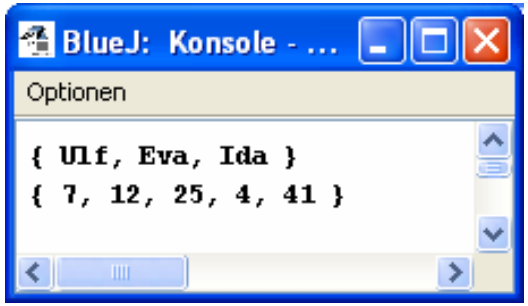
```
public class BSet<E>{
// Klassenfeld
   static int maxSize=100;
// Objektfelder
   /** Menge: s[0 .. füllStand-1] */
   private E[] s;
   private int füllStand;

  /** Konstruiere die leere Menge */
   public BSet(){
      s = (E[])new Object[maxSize];
      füllStand=0;
   }

  /** Konstruktor für nichtleere Menge;*/
   public BSet(E ... elemente)
                        throws SetException{
     this();
     for(E e:elemente){ insert(e); }
     // Prüfe Integrität
     assert nichtsDoppelt();
   }
```

Generische Klassen - Instanziierung

- Benutzung generischer Klassen:
  - Generische Parameter instanziieren
- Nur Objekttypen erlaubt
  - primitive Typen nicht erlaubt
  - hier kein Autoboxing
  - „unexpected Type“

```
public static void genericBoundedSetTest(){

    BSet<String> namen =
          new BSet<String>("Ulf","Eva","Ida");
    System.out.println(namen);

    BSet<Integer> lotto =
          new BSet<Integer>();
    for(int i=0;i<6;i++)
          lotto.insert((int)(Math.random()*49));
    System.out.println(lotto);
}
```

Mit vernünftig implementiertem toString() erhalten wir:



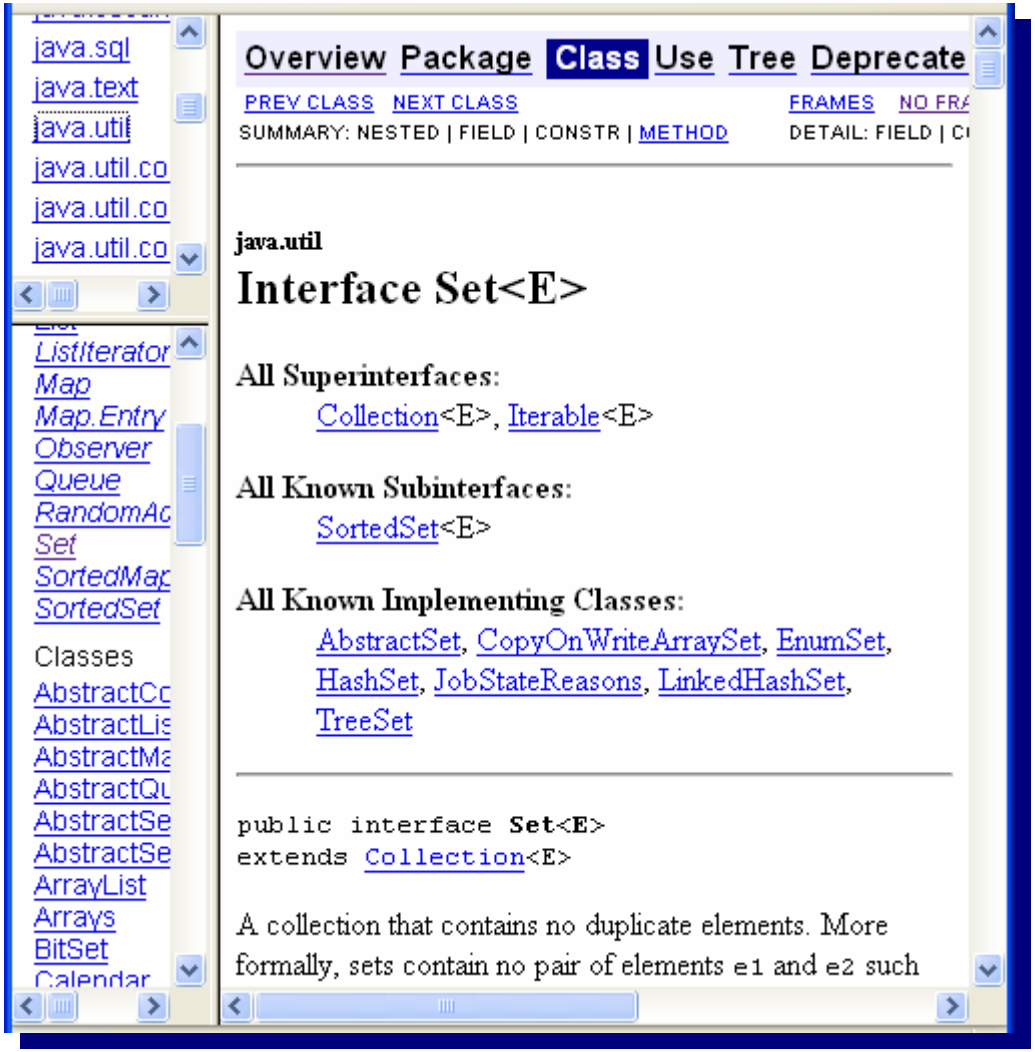
Mehrere generische Parameter

- Eine Klasse für Paare beliebiger Datentypen
- Beispielhafte Instanziierungen
  - Pair<String,Integer> telmey
    = new Pair<String,Integer> („Meyer“,67451);
  - BSet<Pair<String,Integer>> telBuch
    = new BSet<Pair<String,Integer>>
    (telmey,
    new Pair<String,Integer> („Müller“,1234));
  - Pair<Integer,Pair<String,String>> etwas =
    new Pair<Integer,Pair<String,String>>
    (3,new Pair<String,String> („Otto“,“Anna“));
  - System.out.println
    (new Pair<Integer,Integer>(3,4));

```
public class Pair<X,Y>{
   // Objektfelder
   X x;
   Y y;
   // Konstruktoren
   Pair(){}
   Pair(X x, Y y)
      { this.x=x; this.y=y; }

   // toString
   public String toString(){
         return "("+x+","+y+")";
   }
   // equals
   public boolean equals(Pair<X,Y> p){
     return this.x.equals(p.x)
          && this.y.equals(p.y);
   }
```

Sets in *java.util*

- In java.util existiert
  - *interface Set<E>* mit
  - SubInterface *SortedSet<E>*
- Implementierungen u.a.;
  - *HashSet*
  - *TreeSet*
  - *…*



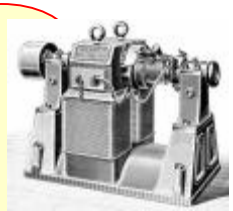
Generatoren

- In BSet<E> können wir
  - *Mengen konstruieren*
  - Elemente aufnehmen
  - *Elemente löschen*
  - *prüfen, ob ein Element in der Menge ist*
- Können wir aufgrund dieser Operationen auch
  - feststellen, ob die Menge leer ist,
  - Anzahl der Elemente zählen,
  - prüfen, M $\subseteq$ N ist oder gar M=N ?
- Können wir
  - M $\cup$ N
  - M $\cap$ N
  - M \N berechnen ?
- Antwort: jein
  - ja, falls E nur endlich viele Element hat
  - nein, sonst
- Was fehlt noch :
  - eine Möglichkeit, die Elemente der Menge der Reihe nach zu produzieren
  - dies nennt man Generator
  - in Java: Iterator

BSet<E>

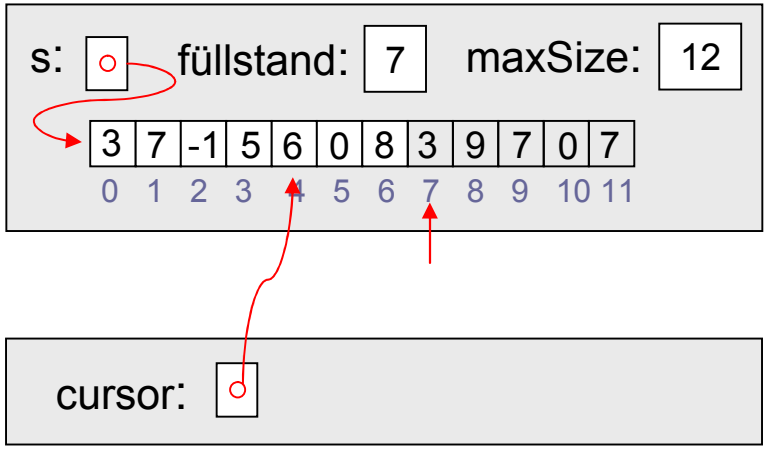
«Konstruktoren»
+ BSet<E> ( )
+ BSet<E>(E … es)
«Mutatoren»
+ void insert(E e)
+ void delete(E e)
«Selektoren/Prädikate»
+ boolean contains(E e)
«Utils»
+ String toString(E e)

«Generator»
+ Generator g ( )



Durchlaufen – die einfache Art

- *cursor* zeigt auf aktuelle Position im Array
  - initCursor()
    setzt cursor auf 0
  - für jedes neu angeforderte Element
    - gib s[cursor] zurück
    - erhöhe cursor
  - Durchlauf beendet, falls cursor == füllStand




```
public void initCursor(){ cursor=0; }

public E next(){ return s[cursor++]; }

public boolean hasNext(){
    return cursor < füllStand;
}
```

Generate-test-collect

- Häufige Aufgabe
  - Gegeben
    - eine Menge M
    - ein Kriterium krit()
    - eine Funktion f: X $\rightarrow$Y
  - berechne { f(x) | x $\in$M $\wedge$ krit(x) }
- Programmierschema
  - Generiere die Elemente der Menge
  - teste jeweils ob das Kriterium erfüllt ist
  - wende die Funktion an
  - gib die Menge der Ergebnisse zurück

Beispiele:

{ x.adresse( ) | x $\in$ kunden $\wedge$ x.konto( ) < 0 }

{ x^2 | x $\in$ {0, ... ,10} && x %3==1 }

```
zahlen.initCursor();
while(zahlen.hasNext()){
   int x = zahlen.next();       // generate
   if(x%3==1)                   // test
        ergebnis.insert(x*x);   // collect
 }
```

siehe:
SetTest.genTestExample()

Genau das hatte uns gefehlt

- Mit den Generatormethoden initCursor(), next(), hasNext() lassen sich alle anderen Operatoren implementieren
  - $M \subseteq N$: durchlaufe M, prüfe jeweils, ob jedes $e \in M$ auch in N ist
- analog
  - $M \cup N$: durchlaufe M und N und sammle die Elemente, die in M oder in N sind
  - $M \cap N$: sammle die $m \in M$, die auch in N sind
  - $M \setminus N$: sammle die $m \in M$, die nicht in N sind

```
public BSet<E> union(BSet<E> mengeN){
    BSet<E> union = new BSet<E>();
    // Lege eine Kopie von this an
    this.initCursor();
    while(this.hasNext()) union.insert(this.next());
    // füge alle Elemente von mengeN hinzu
    mengeN.initCursor();
    while(mengeN.hasNext())union.insert(mengeN.next());
    return union;
}
```

BSet<E>

«Konstruktoren»
+ BSet<E> ( )
+ BSet<E>(E … es)

«Mutatoren»
+ void insert(E e)
+ void delete(E e)

«Selektoren/Prädikate»
+ boolean contains(E e)

«Utils»
+ String toString(E e)

«Generatormethoden»
+ E next ( )
+ boolean hasNext()
+ void initCursor()

Behälter und Iteratoren

- Bisherige Behälter für Datenobjekte:
  - *Listen, Mengen, Arrays, Stacks, Queues*
- Häufige Aufgabe: *Iterieren:*
  - Durchlaufe alle Objekte x in Behälter b und tue was damit
    - Der Durchlauf soll b nicht verändern
- Pseudocode:
  - *Für alle* x *in* b : tueWas(x);
- Verfeinerung
  - while (**?**.hasNext()){
    - x = **?**.next();
    - tueWas(x);

    }
- Was könnte **?** sein ?
  - ? = b : geht nicht, b soll sich ja nicht verändern
  - ? muss ein **Iterator** (Durchläufer) für b sein.
- Ein Iterator ist ein Objekt
  - es kann initialisiert werden
  - es braucht Methoden
    - boolean **hasNext**()
    - Object **getNext**();

<<interface>>

Iterator<E>

«methods»

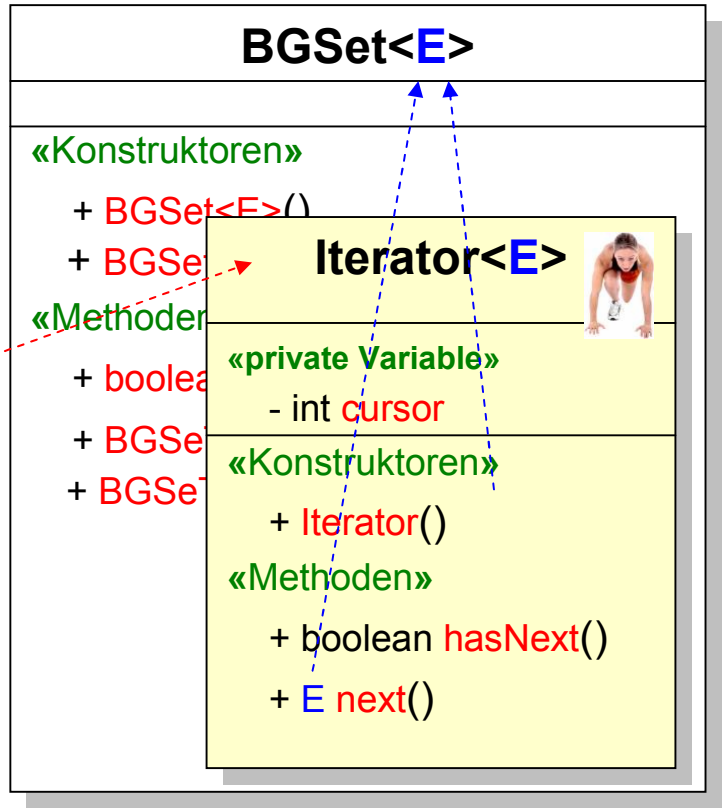
+ boolean hasNext()
+ E next( )
+ void remove(E e)

interface: java.util.Iterator

Mehrere Generatoren/Iteratoren

- das folgende Beispiel geht nicht:
  - $\{ ||x^2-y^2|| \mid x, y \in M \wedge x \neq y \}$
- Problem : zweiter unabhängiger Generator für y benötigt
  - aber jede *BSet* besitzt nur eine Variable cursor
- Lösung: *Abstraktion*

  Innere Klasse *Iterator* kapselt
  - *Initialisierung* (Konstruktoraufruf)
  - *Überprüfung* (*public hasNext()*)
  - *Erzeugung* (*public next()*)
- new Iterator
  - erzeugt frischen Iterator
  - mit privatem cursor



Interface *java.util.Iterator<E>*

- Ein Iterator soll der Reihe nach alle Elemente eines Behälters produzieren

- er muss jeweils
  - prüfen können, ob alle Elemente besucht wurden
    - *hasNext()*
  - das nächste Element liefern
    - *next()*
  - ggf. das gegenwärtige Element entfernen
    - *remove()*

- *remove()* wird oft durch *UnsupportedOperationException* implementiert

Method Summary

boolean	**hasNext**() Returns `true` if the iteration has more elements.
E	**next**() Returns the next element in the iteration.
void	**remove**() Removes from the underlying collection the last element returned by the iterator (optional operation).

Private Generatorklasse in BGSet

- Wichtig sind
  - Konstruktor *Generator()*
  - *next()*
  - *hasNext()*
- Interface verlangt noch
  - *remove()*
- Was bietet das Interface ?
  - *foreach Syntax ?*
  - *.... abwarten*

BGSet

Klasse Bearbeiten Werkzeuge Optionen

Übersetzen | Rückgängig | Ausschneiden | Kopieren | Einfügen | Suchen... | Weitersuchen | Schließen

```
private class Generator implements Iterator<E>{
    // Instanzvariable
    private int cursor;
    // Konstruktor
    public Generator(){ cursor=0; }
    // die drei Methoden
    public E next() { return s[cursor++]; }

    public boolean hasNext(){
        return cursor < füllStand;
      }
    // diese wird oft nur so implementiert:
    public void remove(){
      throw new UnsupportedOperationException();}
  }
```

Generatoren für jede Menge

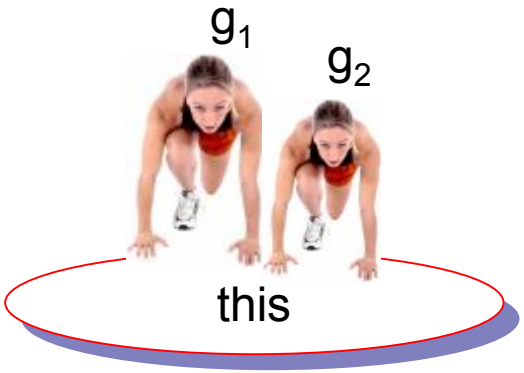
- Generatoren sind bis jetzt nur dem Objekt *this* zugeordnet
  - Programmierung von *union, product* etc. noch nicht möglich
  - wir benötigen auch Generator für die Argumentmenge
  - *mengeM.union(mengeN)*
- Lösung
  - Methode *Iterator iterator( )*
  - liefert einen *Iterator* für jede Menge


```
public BGSet<E> union(BGSet mengeN){
   // Menge für das resultat
   BGSet<E> union = new BGSet();
   Generator g1 = new Generator();
   // ... und wie gewinnen wir für
   // mengeN auch einen Generator  ?
   // ....  ??? .....
```

```
  public Iterator<E> iterator(){
      return new Generator();
  }
```

```
public BGSet<E> union(BGSet<E> mengeN){
   BGSet<E> union = new BGSet();

   Iterator<E> g1 = this.iterator();
   while(g1.hasNext()) union.insert(g1.next());

   Iterator<E> g2 = mengeN.iterator();
   while(g2.hasNext()) union.insert(g2.next());

   return union;
} // end of union
```

Zwei Generatoren auf einer Menge



- Codestruktur analog zu geschachtelten *for*-Schleifen

BGSet

Klasse Bearbeiten Werkzeuge Optionen

Übersetzen | Rückgängig | Ausschneiden | Kopieren | Einfügen | Suchen... | Weitersuchen | Schließen | Implementie

```
public BGSet<Pair<E,E>> square(){
    // Menge für das resultat
    BGSet<Pair<E,E>> square = new BGSet<Pair<E,E>>();

    Generator g1 = new Generator();
    while(g1.hasNext()){
       E x = g1.next();

       Generator g2 = new Generator();
       while(g2.hasNext()){
          E y = g2.next();
          square.insert(new Pair<E,E>(x,y));
       }
    }
    return square;
}
```

`new BGSet<String>("Anna","Eva","Ulf").square().toString()` liefert:

BlueJ: Konsole - abstrakteDatentypen

Optionen

```
{ (Anna,Anna), (Anna,Eva), (Anna,Ulf), (Eva,Anna), (Eva,Eva), (Eva,Ulf), (Ulf,Anna), (Ulf,Eva), (Ulf,Ulf) }
```

Iterator in Java

- Wir verpassen jedem Behältertyp
  - Arrays, Listen, Mengen, …
- eine Methode
  - *Iterator iterator()*
- Ist b ein Behälter, so liefert
  - *Iterator guide = b.iterator();*
- einen Führer durch den Behälter.

- Programme, um Listen oder Mengen zu durchlaufen unterscheiden sich nicht

Set

«Konstruktor»
+ SetOfObjects( )
«updates»
+ void insert (*Object*)
+ void delete(Object)
....
«iterators»
+ Iterator iterator( )

Liste

«Konstruktor»
+ Liste( )
«updates»
+ void einfügen (*Object*)
+ void entfernen(Object)
....
«iterators»
+ Iterator iterator( )

```
import java.util.Iterator;
public class Set{
// Private Felder
  private Object[] theSet;
  private int füllStand=0;

  Iterator iterator(){
    return new SetIterator();
  }

  private class SetIterator implements Iterator{
    int zeiger;

    SetIterator(){ int zeiger = 0;}

    public boolean hasNext(){
      return zeiger < füllStand;
    }
    public Object next(){
      return theSet[zeiger++];
    }
    public void remove(){
      theSet[zeiger] = theSet[--füllStand];
    }
```

Iterator-Anwendung

- Wir basteln eine kleine Menge
- besorgen uns einen Führer, der uns durch die Menge führt
- Um alle Paare von Elementen zu finden
  - besorgen wir uns einen Führer a.
  - Für jedes neue Element, das er uns zeigt, besorgen wir einen Führer b.
- Wir drucken die gefundenen Elemente

```
Set menge = new Set(10);
menge.insert("Otto");
menge.insert("Eva");
menge.insert("Anton");

/** Drucke alle Elemente aus */
Iterator führer = menge.iterator();
while(führer.hasNext()){
  System.out.println(führer.next());
}

/** Drucke alle Paare von Elementen */
Iterator a = menge.iterator();
while (a.hasNext()){
  Object x = a.next();
  Iterator b = menge.iterator();
  while (b.hasNext()){
      Object y = b.next();
      System.out.println("("+x+","+y+")");
  }
}
```

BlueJ: Terminal...

Options

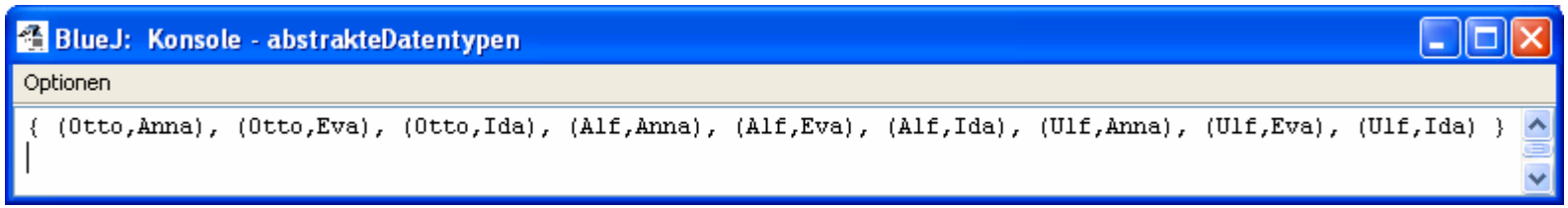
```
Otto
Eva
Anton
(Otto,Otto)
(Otto,Eva)
(Otto,Anton)
(Eva,Otto)
(Eva,Eva)
(Eva,Anton)
(Anton,Otto)
(Anton,Eva)
(Anton,Anton)
```

Das interface *Iterable*

- verlangt nur eine Methode:
  - *Iterator iterator()*
- schenkt uns *foreach-Syntax*, um Behälter *beh* zu durchlaufen
  - sofern diese Iterable implementieren
- for(E e : *beh*) {
  ... tu was mit e ...
  }
- Iteratoren/Generatoren werden gar nicht mehr erwähnt
  - aber sie bringen die Sache zum Laufen

```
import java.util.Iterator;
public class BGSet<E> implements Iterable<E>{
```

```
public BGSet<Pair<E,E>> product(BGSet<E> mengeN){
    BGSet<Pair<E,E>> product = new BGSet<Pair<E,E>>();
    for(E x: this)
       for(E y : mengeN)
          product.insert(new Pair<E,E>(x,y));
   return product;
}// end of product
```



java.lang.*Iterable*

- Verlangt nur eine Methode
  Iterator<E> iterator()
- Schenkt *foreach*-Syntax
- Viele bekannte Klassen implementieren *Iterable*
- Wichtigstes Subinterface:
  - *Collection<E>*
- *Collection<E>* Objekte dienen als Behälter für Objekte vom Typ E
- Beispiele
  - Mengen
  - Listen
  - Stacks
  - Maps
  - Bäume
  - ...

Das Collection Interface

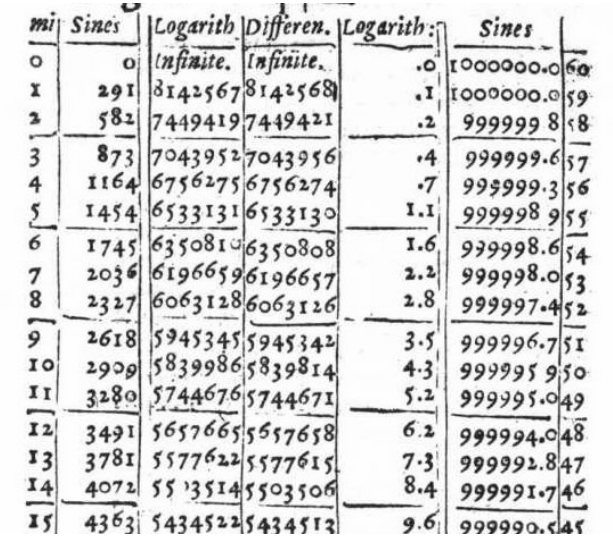
- Elemente sind vom Typ *Object*
- boolesche Operationen add..., remove... liefern true, falls sie erfolgreich waren (Behälter wurde verändert)
- Test auf Gleichheit von Elementen durch equals
- „*Bulk*“- Operationen löschen/addieren einen kompletten Behälter
- retainAll entfernt alle außer denen im Argument-Behälter (Schnitt)
- toArray kippt Behälter in einen Array aus
- iterator liefert einen Iterator durch die Elemente des Behälters

```
public interface Collection {

// Basic Operations
  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(Object element); // Optional
  boolean remove(Object element); // Optional
  Iterator iterator();

// Bulk Operations
  boolean containsAll(Collection c);
  boolean addAll(Collection c); // Optional
  boolean removeAll(Collection c); // Optional
  boolean retainAll(Collection c); // Optional
  void clear(); // Optional

// Array Operations
  Object[] toArray();
}
```

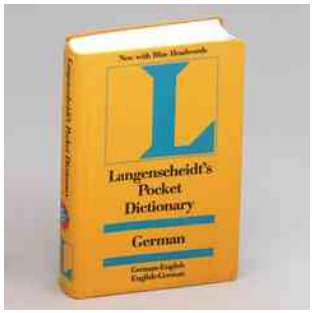
Überlegen Sie:
Wie muss **Collection<E>**
in Java 1.5 aussehen ?

Maps: Tabellen, Wörterbücher, Indices

- Map = Abbildung
- Zuordnung:
  - Argument |→ Wert
- Zu jedem Argument gehört höchstens ein Wert
- Anwendungen
  - Tabellen
  - tabellierte Funktionen
    - Logarithmentabelle
    - Trig.Tabelle
  - Wörterbücher
  - Index
  - PLZ-Buch
  - Suchmaschinen



mi	Sines	Logarith	Differen.	Logarith:	Sines	
0	0	Infinite.	Infinite.	.0	1000000.0	60
1	291	8142567	8142568	.1	1000000.0	59
2	582	7449419	7449421	.2	999999 8	58
3	873	7043952	7043956	.4	999999.6	57
4	1164	6756275	6756274	.7	999999.3	56
5	1454	6533131	6533130	1.1	999998 9	55
6	1745	6350810	6350808	1.6	999998.6	54
7	2036	6196659	6196657	2.2	999998.0	53
8	2327	6063128	6063126	2.8	999997.4	52
9	2618	5945345	5945342	3.5	999996.7	51
10	2909	5839986	5839814	4.3	999995 9	50
11	3280	5744676	5744671	5.2	999995.0	49
12	3491	5657665	5657658	6.2	999994.0	48
13	3781	5577622	5577615	7.3	999992.8	47
14	4072	55 3514	5503506	8.4	999991.7	46
15	4363	5434522	5434513	9.6	999990.5	45



033	!	066	B	099	c	132	ä
034	"	067	C	100	d	133	à
035	#	068	D	101	e	134	å
036	$	069	E	102	f	135	ç
037	%	070	F	103	g	136	ê
038	&	071	G	104	h	137	ë
039		072	H	105	i	138	è
040	(	073	I	106	j	139	ï
041	)	074	J	107	k	140	î
042	*	075	K	108	l	141	ì
043	+	076	L	109	m	142	Ä

aus: Einführung. in die Informatik

Datenstruktur : Map / Dictionary

- Grundmenge: $Map \subseteq Keys \times Values.$
  - Operationen
    - $newMap : \rightarrow Map$
    - $insert : Map \times Keys \times Values \rightarrow Map$
    - $remove: Map \times Keys \rightarrow Map$
    - $isEmpty : Map \rightarrow boolean$
    - $containsKey : Map \times Keys \rightarrow boolean$
    - $lookup : Map \times Keys \rightarrow Value$
  - Gleichungen
    - lookup(insert(m,k,v),k) = v
    - lookup(insert(m,k_2,v), k_1) = lookup(m,k_1), falls $k_1 \neq k_2$
      etc.
  - Mathematisch:
    - lookup = Anwendung der Abbildung auf ein Argument:
      f(x) = lookup(f,x)

Map

- Eine Map ordnet
  - einem Argument (key)
  - einen Wert (value) zu
- Eine Map entsteht durch
  - einfügen (insert)
  - löschen (remove)

  eines (key,value)-Paares
- Selektoren:
  - containsKey
  - lookup
- Meist will man containsKey mit lookup verbinden:
  - Vereinbare „Sentinel“-Wert (z.B. null):
    containsKey(k) ⇔ lookup(k) != null

Map<KTyp,VTyp>

«updates»
+ void insert (KTyp, VTyp)
+ void remove (KTyp)

«query»
+ boolean containsKey (KTyp)
+ VTyp lookup (KTyp)

Einträge - Entries

- Ein Eintrag (entry) ist ein Paar (k,v) bestehend aus
  - einem Schlüssel (*key*)
  - einem Wert (*value*)
- Kein Setter außer dem Konstruktor
- *toString* und *equals* überschreiben

```
public class Entry<KTyp,VTyp>{
// Objektfelder
   KTyp     key;
   VTyp value;
// Konstruktor
   Entry(KTyp k,VTyp v){
      key = k;
      value = v;
   }
// getter
   KTyp getKey()  { return key; }
   VTyp getValue(){ return value; }

// overriding
   public String toString(){
     return "( "+key+" --> "+value+" )";
   }
   public boolean equals(Object o){
     return this.key.equals(((Entry<KTyp,VTyp>)o).key)
       && this.value.equals(((Entry<KTyp,VTyp>)o).value);
   }
}
```

Map ist Menge von Entries

Map

Klasse Bearbeiten Werkzeuge Optionen

Übersetzen | Rückgängig | Ausschneiden | Kopieren | Einfügen | Suchen... | Weitersuchen | Schließen | Implementieru...

```
public class Map<KTyp,VTyp> extends BGSet<Entry<KTyp,VTyp>>{

   Map(){ super(); }

   /* Klasseninvariante garantiert Map-Eigenschaft:
    * - zu jedem Schlüssel (key)
    * - gibt es höchstens einen Wert (value) */
   private boolean invariant(){
      for (Entry<KTyp,VTyp> e : this)
        for (Entry<KTyp,VTyp> f : this)
          if (e.getKey().equals(f.getKey()))
            if ( ! e.getValue().equals(f.getValue()))
              return false;
      return true;
```

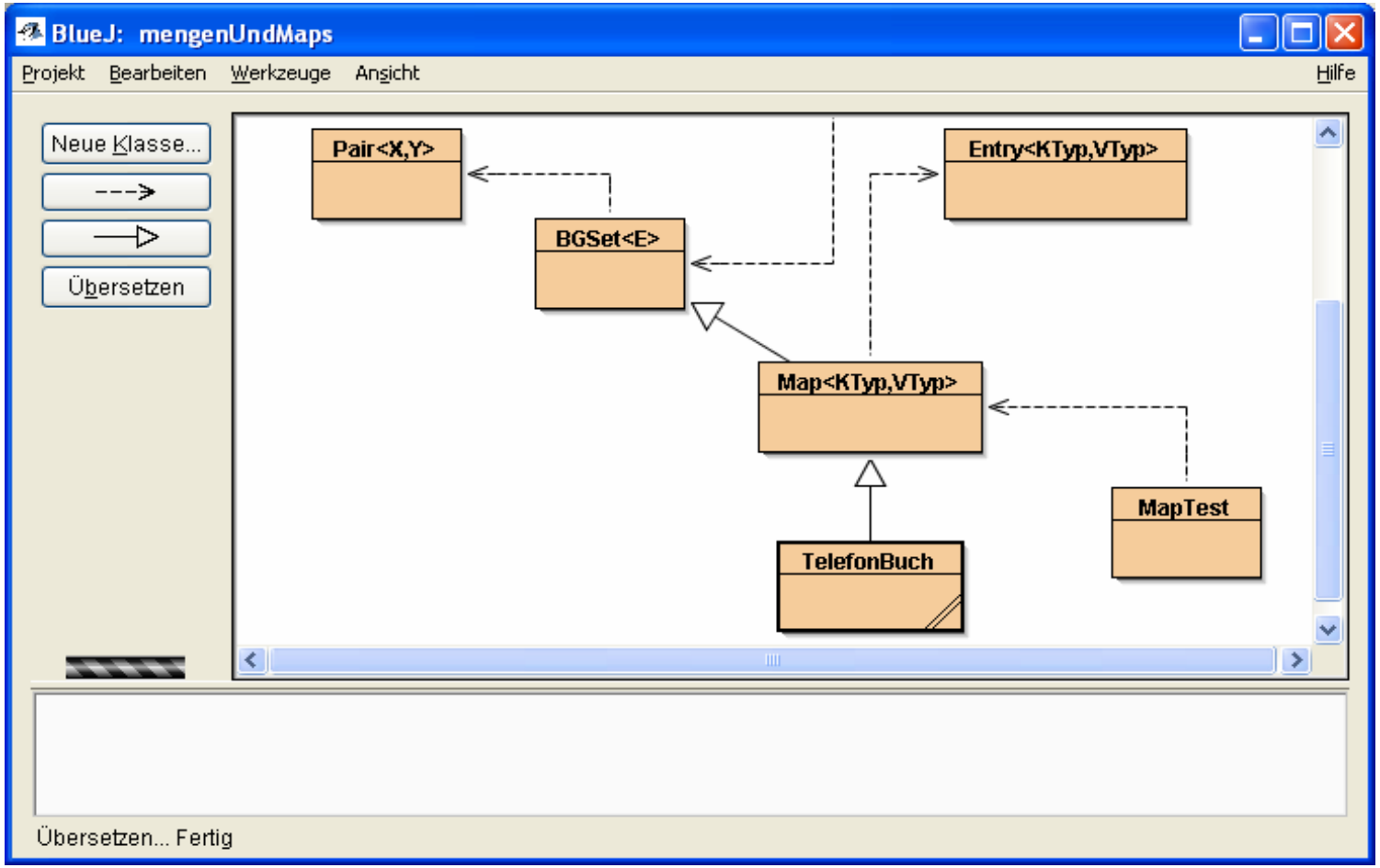
geändert

- Abbildungs-eigenschaft:

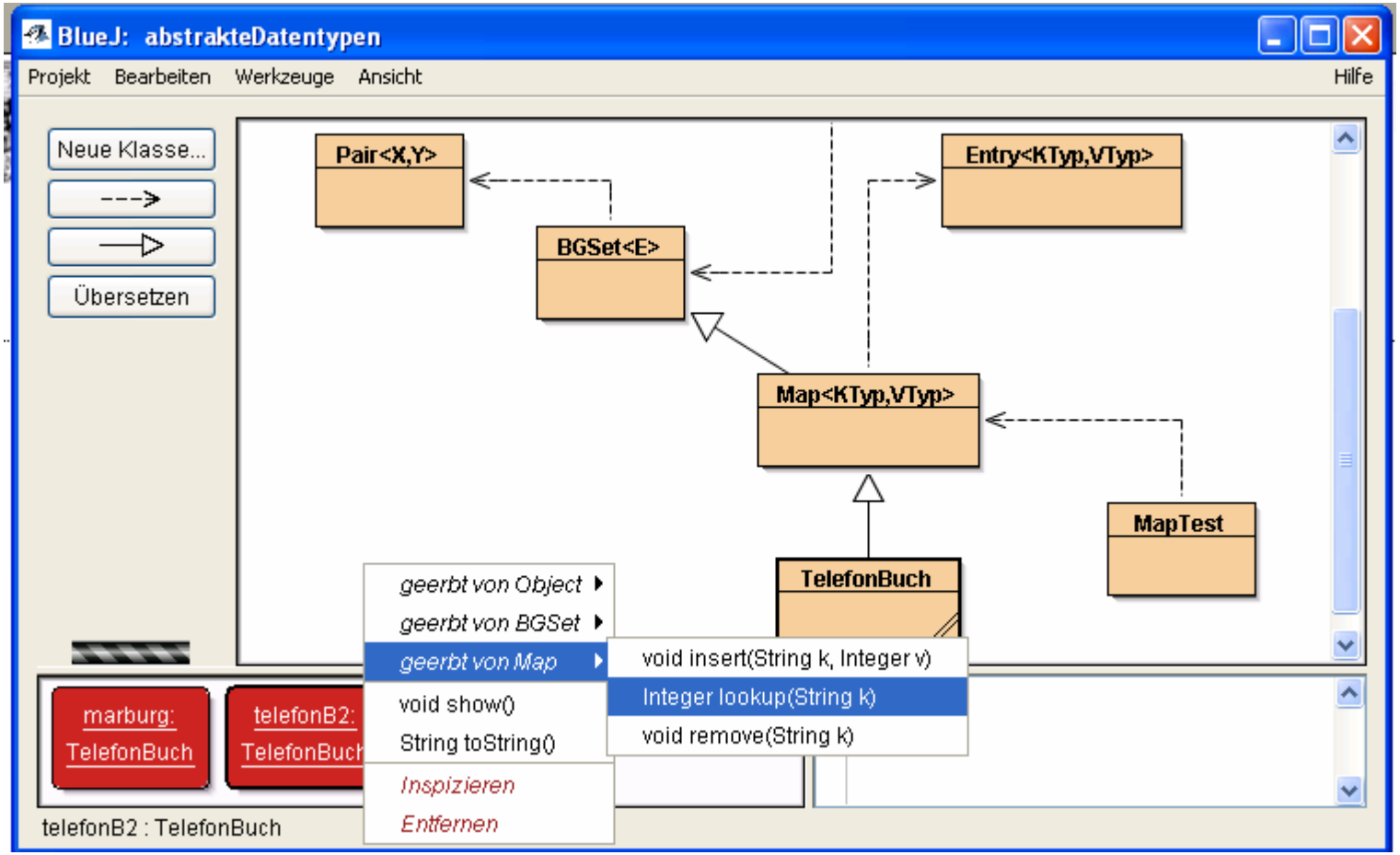
$\forall\ (e,v_1), (f,v_2) \in$ Map:

$e = f \Rightarrow v_1 = v_2$

Ein Telefonbuch ist eine Map



Ein Telefonbuch ist eine Map



Ein Telefonbuch ist eine Map