



Einfache Datenstrukturen

Abstrakte Datentypen
Behälter, Mengen, Stacks,
Queues, Anwendungen



Einheit von Daten und Operationen

- Daten und Operationen auf Daten gehören zusammen
 - Konto:
 - Daten:
 - name, nummer, kontoStand
 - Operationen
 - überweisen, abheben, ...
 - Datum
 - Daten:
 - tag, monat, jahr
 - Operationen
 - istSchaltjahr, addiereTage, ...
 - Student
 - Daten:
 - name, geburtsDatum, semester
 - Operationen
 - scheinAusstellen, einschreiben, ...
- Programmierer sollte dies beherzigen
 - Modellierung
- Programmiersprache muss das unterstützen
 - Klassen, Module, Units





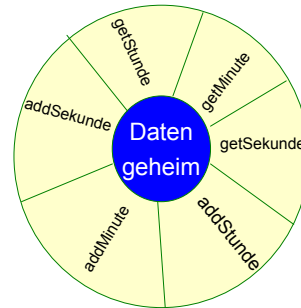
Datenkapselung

- Daten sind wichtig, aber
 - sie sollen nur kontrolliert geändert werden
 - Kontostand nur durch
 - Überweisen, abheben
 - Datum nur durch
 - tage addieren, monat setzen
 - Erzeugung
 - Konstruktor
 - Ablesung
 - Selektoren
 - Veränderung
 - Mutatoren
- Methoden bilden Kapsel um die Daten
 - Repräsentation im Inneren versteckt
 - Class Konto{


```
private int kontoStand;
```
 - Class Datum{


```
private int msecSeit1970;
```
 - Class Student{


```
private Date geburtsDatum;
              private String name, vorname;
```
 - Repräsentation kann man nachträglich ändern, ohne dass der Anwender etwas merkt



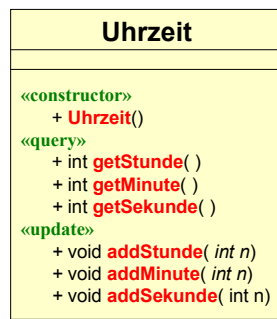
Class UhrZeit

Ob Uhrzeit intern als
mSec seit Mitternacht
oder als Tripel
(h,m,s)
gespeichert wird ist egal



Benutzersicht - Programmiersicht

- Klassendiagramme
 - Signaturen aller Methoden
 - + falls public
 - - falls private
- Wichtig für
 - Entwurf
 - Implementierung
 - Dokumentation
- Zusätzliche Spezifikation i.A. notwendig



Klassendiagramm

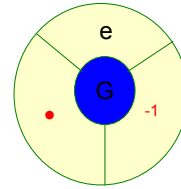
Implementierung

```
public class Uhrzeit{
    public Uhrzeit(...){
        ...
    }
    public int getStunde(){
        ...
    }
}
```



Daten und Operationen in der Mathematik

- Aus der Mathematik kennen wir
 - Ein Gruppe ist eine Menge G mit Operationen
 - $\cdot : G \times G \rightarrow G$
 - $^{-1} : G \rightarrow G$
 - $e : \rightarrow G$
 die gewisse Gleichungen erfüllen ..
 - Eine Boolesche Algebra ist eine Menge B mit Operationen
 - $\wedge, \vee : B \times B \rightarrow B$
 - $\neg : B \rightarrow B$
 - $0, 1 : \rightarrow B$
 die gewisse Gleichungen erfüllen
 - Ein Körper ist eine Menge K mit Operationen
 - $+, \cdot, ^{-1} : K \times K \rightarrow K$
 - $0, 1 : \rightarrow K$
 Wobei gewisse Gleichungen erfüllt sein müssen.
Die Operation $^{-1}$ ist partiell – d.h. nicht überall definiert.
 - Ein Vektorraum über einem Körper K ist eine Menge V mit Operationen
 - $+, \cdot : V \times V \rightarrow V$
 - $0 : \rightarrow V$
 - $\cdot : K \times V \rightarrow V$
- Solche mathematischen Gebilde, bestehend aus einer oder mehreren Mengen und Operationen und evtl. noch Gleichungen, nennt man **Algebren**.



Wichtige Algebren der Informatik

- **Boolesche Algebra**
 - Grundmenge $\{t, f\}$
 - Operationen
 - $\wedge, \vee : B \times B \rightarrow B$
 - $\neg : B \rightarrow B$
 - $t, f : \rightarrow B$
 - Gleichungen
 - $\neg(x \vee y) = \neg x \wedge \neg y$
 - ... etc.



George Boole

Boolesche Algebra
«Konstruktoren» $t, f : \rightarrow B$
«Operationen» $\wedge, \vee : B \times B \rightarrow B$ $\neg : B \rightarrow B$
«Gleichungen» $\neg(x \vee y) = \neg x \wedge \neg y$... etc.



Wichtige Algebren der Informatik

- Die **natürlichen Zahlen** N
 - Grundmenge $\{0, 1, 2, 3, 4, \dots\}$
 - Operationen
 - $0 : \rightarrow N$
 - $\text{succ} : N \rightarrow N$
 - $+$: $N \times N \rightarrow N$
 - $\text{div}, \text{mod} : N \times N \rightarrow N$
 - $=$: $N \times N \rightarrow B$
 - Gleichungen
 - $x+0=x$
 - $x+\text{succ}(y) = \text{succ}(x+y)$
 - etc.



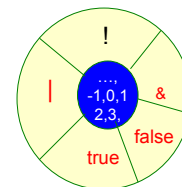
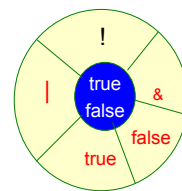
Giuseppe Peano

Natürliche Zahlen	
«Konstruktoren»	
0	$: \rightarrow N$
succ	$: N \rightarrow N$
«Operationen»	
$+$	$: N \times N \rightarrow N$
div, mod	$: N \times N \rightarrow N$
$=$	$: N \times N \rightarrow B$
«Gleichungen»	
$(x + 0) = x$	
$(x + \text{succ}(y)) = \text{succ}(x+y)$	
... etc.	



Realisierung in Programmiersprachen

- In **Java**
 - Datentyp **boolean**
 - Wertmenge $\{\text{true}, \text{false}\}$
 - Operationen
 - $\&, |$: $\text{boolean} \times \text{boolean} \rightarrow \text{boolean}$
 - $!$: $\text{boolean} \rightarrow \text{boolean}$
 - $\text{true}, \text{false}$: $\rightarrow \text{boolean}$
 - zusätzlich
 - $\&\&, ||, \wedge$: $\text{boolean} \times \text{boolean} \rightarrow \text{boolean}$
- In **C, C++** :
 - **Alle ganzen Zahlen** sind boolesche Werte.
 - Bedeutungsfunktion $\varphi(x) := x==0$, d.h.
 - $=0$: true
 - $\neq 0$: false
 - $\&, |, !$ auf alle Zahlen anwendbar. Damit das „gut geht“, muss :
 - $\varphi(x \& y) = \varphi(x) \wedge \varphi(y)$
 - $\varphi(x | y) = \varphi(x) \vee \varphi(y)$
 - $\varphi(!x) = \neg \varphi(x)$
 - Wie kann man $\&, |, !$ auf int definieren, so dass dies gilt.
 - **Nachteil:** Dumme Fehler werden nicht als solche erkannt
 - `if (x >= 'a' + x <= 'z') printf("Kleinbuchstabe");`





N in Programmiersprachen

■ Natürliche Zahlen

□ In PASCAL:

0 1 2 3 4 5 6 7 8 ...

- CARDINAL \approx natürliche Zahlen
 - Ist n CARDINAL, dann ist -n ein Fehler.
- Nur ein endlicher Bereich vorhanden
 - $[0 .. 2^n - 1]$
 - Häufig : n=32 oder n=64
 - Für Bereichsüberschreitung ist der Programmierer verantwortlich

□ In Java

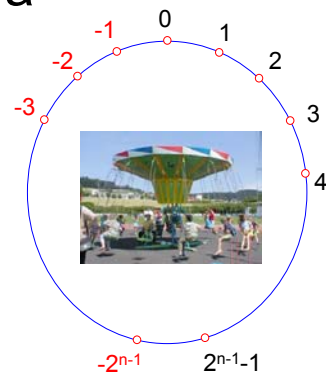
- Nicht gesondert vorhanden
- natürliche Zahlen sind ja enthalten in den ganzen Zahlen
 - Verwende nur die positiven Werte vom Typ byte, short, int, long
 - $[0 .. 2^n - 1]$, n=8,16,32,64.
- Nachteil:
 - siehe vorige Folie

... ~~8-7-6-5-4-3-2-1~~ 0 1 2 3 4 5 6 7 8 ...



Ganze Zahlen in Java

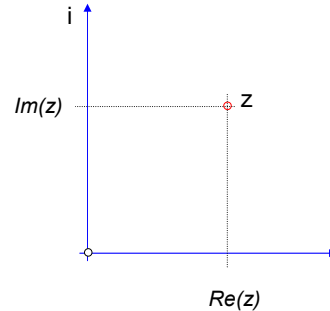
- 4 Angebote
 - byte, short, int, long
- Jeweils nur ein Intervall der Form $[-2^{n-1} .. 2^{n-1}-1]$
 - N=8,16,32, oder 64
- Zusätzliche Operationen, die mathematisch kaum relevant sind
 - $\&$, $|$, \sim (bitweise Boolesche Operationen),
 - \ll , \gg (left, right shift)
 - \ggg (right shift with zero)
- Sofern Operationen nicht das Intervall verlassen, sind sie exakt
 - Viele Operationen arbeiten in Wirklichkeit modulo 2^n
- Auch Zwischenergebnisse dürfen das Intervall nicht verlassen.
- Diese Einschränkung gilt auch für Gleichungen
 - Beispiel: $x-y+z \equiv x+z-y$ gilt nicht unbedingt ! Warum ?





Die komplexen Zahlen

- Wichtig in Physik und E-Technik
 - Grundmenge $C = R \times R = \{ (x,y) \mid x,y \in R \}$
 - Statt (x,y) schreibt man auch $x+iy$
 - Operationen
 - $+$: $C \times C \rightarrow C$
 - $*$: $C \times C \rightarrow C$
 - $|\cdot|$: $C \rightarrow R$
 - Re, Im : $C \rightarrow R$
 - Gleichungen
 - $(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$
 - $(x_1, y_1) * (x_2, y_2) = (x_1 * x_2 - y_1 * y_2, x_1 * y_2 + y_1 * x_2)$
 - $Re(x, y) = x$
 - $Im(x, y) = y$
 - Reelle Zahlen r, s kann man mit den komplexen Zahlen $(r, 0)$, bzw $(s, 0)$ identifizieren.
 - Das geht, weil diese Identifizierung die Operationen respektiert:
 - $(r, 0) * (s, 0) = (r * s, 0)$
 - $(r, 0) + (s, 0) = (r + s, 0)$
- Die komplexen Zahlen sind in Java nicht als Datentyp vorhanden
 - Wir müssen ihn selber implementieren



Spezifikation

- Grundmenge: Complex
 - $R \times R$
- Operationen:
 - $+$: $Complex \times Complex \rightarrow Complex$
 - $*$: $Complex \times Complex \rightarrow Complex$
 - Re : $Complex \rightarrow R$
 - Im : $Complex \rightarrow R$
- Gleichungen:
 - Definierende Gleichungen für add, mult, Re, Im
 - $(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$
 - $(x_1, y_1) * (x_2, y_2) = (x_1 * x_2 - y_1 * y_2, x_1 * y_2 + y_1 * x_2)$
 - $Re(x, y) = x$
 - $Im(x, y) = y$
 - Zusätzliche Gleichungen analog wie für R .

Komplexe Zahlen
«Konstruktoren» + Complex() + Complex(float, float)
«Operationen» + Complex sum(Complex, Complex) + Complex prod(Complex, Complex)
«Selektoren» + float getRealteil(Complex) + float getImaginärteil(Complex) + String toString(Complex)



Implementierung in Java

- Datentypen modelliert durch Klassen

- Grundmenge:
 - Kombination von Feldern
- Operationen
 - Methoden

- Hier:

- Grundmenge
 - float × float
- Re
 - getRealteil
- Im
 - getImaginärteil

```
public class Complex
{
    // Eine komplexe Zahl ist ein Paar von reellen Zahlen
    float x; // Realteil
    float y; // Imaginärteil

    // Konstruktoren
    public Complex() {}

    public Complex(float realteil, float imaginärteil) {
        x = realteil;
        y = imaginärteil;
    }

    // Selektoren
    public float getRealteil() {
        return x;
    }

    public float getImaginärteil() {
        return y;
    }
}
```



Operationen - statische Methoden

- In der Darstellung der Operation

- + : $Complex \times Complex \rightarrow Complex$
- sind beide Argumente gleichberechtigt.
- dies modellieren wir, wenn wir add als Klassenmethode (static) implementieren:



```
/** Komplexe Addition */
public static Complex sum(Complex z1, Complex z2) {
    return new Complex(z1.getRealteil()+z2.getRealteil(),
        z1.getImaginärteil()+z2.getImaginärteil());
}

/** Komplexe Multiplikation */
public static Complex prod(Complex z1, Complex z2) {
    float x1 = z1.getRealteil();
    float y1 = z1.getImaginärteil();
    float x2 = z2.getRealteil();
    float y2 = z2.getImaginärteil();
    return new Complex(x1*x2-y1*y2, x1*y2+y1*x2);
}

//Darstellung
public String toString() {
    if (y==0) return ""+x;
    else return ""+x+ " + i("+y+" )";
}
```



Ein Testaufruf

- Statische Methoden
 - wie sum, prod
- sind Methoden der Klasse.
- Ein Aufruf verlangt als Adressat der Methode den Namen der Klasse:
 - `Complex.add(z1,z2)`
- Bei dem Aufruf aus der definierenden Klasse, kann der Adressat entfallen.
 - `prod(z1,z2)`

The screenshot shows the BlueJ IDE interface. On the left, a code editor displays the following Java code:

```
public static void test(float x1, float y1, float x2, float y2){
    Complex z1 = new Complex(x1,y1);
    Complex z2 = new Complex(x2,y2);

    System.out.print("Summe ist ");
    System.out.println(Complex.sum(z1,z2));
    System.out.print("Produkt ist ");
    System.out.println(prod(z1,z2));
}
```

In the center, a terminal window titled "BlueJ: Terminal Window" shows the output:

```
Summe ist 0.0 + i(2.0)
Produkt ist -1.0
```

On the right, a "BlueJ: Method Call" dialog box is open, showing the method signature `void test(float x1, float y1, float x2, float y2)` and input fields for the arguments: `Complex.test(0, 1, 0, 1)`, where the values correspond to the test method's parameters.



Erstes Argument als Empfänger

- Bei einer **Objektmethode** ist das Objekt, nicht die Klasse, der Adressat der Methode.
 - Die **Operationen**
 - `+` : `Complex × Complex → Complex`
 - `*` : `Complex × Complex → Complex`
- werden als **Objekt-Methoden** mit der Signatur
- `Complex plus(Complex z)`
 - `Complex mal(Complex z)`
- implementiert.
- Das erste Argument der Operation ist Empfänger der Methode.
 - `this`
 - Ein beispielhafter **Aufruf** :

Komplexe Zahlen

«Konstruktoren»

- + `Complex()`
- + `Complex(float, float)`

«Operationen»

- + `Complex plus(Complex)`
- + `Complex mal(Complex)`

«Selektoren»

.....

```
/** Komplexe Addition - als Objektmethode*/
public Complex plus(Complex z2){
    return new Complex(x + z2.getRealteil(),
                      y + z2.getImaginarteil());
}

public void testeObjektmethode(){
    Complex i = new Complex(0,1);
    Complex drei = new Complex(3,0);
    System.out.println(drei.plus(i.mal(i)));
}
```



Implementierung als Anweisung

Die Operationen

- $+$: *Complex* \times *Complex* \rightarrow *Complex*
- $*$: *Complex* \times *Complex* \rightarrow *Complex*

werden jetzt als **Anweisungen** implementiert:

- Berechne Ergebnis
- Speichere es im Empfänger

dies entspricht etwa

- $x = x + y$
- $x = x * y$

```

/** Komplexe Addition - als Anweisung*/
public void add(Complex z2) {
    x = x + z2.getRealteil();
    y = y + z2.getImaginarteil();
}

/** Komplexe Multiplikation - Anweisung */
public void mult(Complex z2) {
    float x2 = z2.getRealteil();
    float y2 = z2.getImaginarteil();
    float xNeu=x*x2-y*y2;
    y = x*y2+y*x2;
    x = xNeu;
}

/* ein Test */
public static Complex wasKommtRaus() {
    Complex z = new Complex(1, 1);
    Complex i = new Complex(0, 1);
    z.mult(i);
    z.add(z);
    return(z);
}

```

Komplexe Zahlen
«Konstruktoren»
...
«Operatoren»
+ void add(Complex)
+ void mult(Complex)
«Selektoren»
...



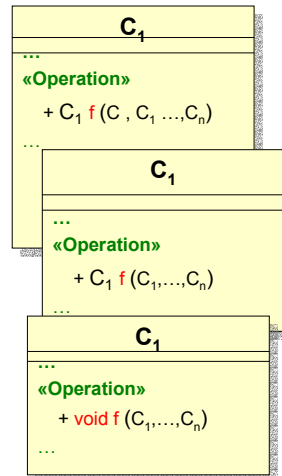
Zusammenfassung

- Einen **Datentyp C** implementiert man in Java als **Klasse C**.
- Eine Operation vom Typ

$$f : C \times C_1 \dots C_n \rightarrow C$$

kann man in einer **Klasse C** implementieren als

- Klassen-Methode** mit Java-Signatur
 - `static C f(C, C1, ..., Cn)`
 - keine Betonung irgendeines Arguments
 - Typischer Aufruf ist ein **Ausdruck** (expression)
 - `x = Klasse.f(x, x1, ..., xn);`
- Objektmethode** in der Klasse C mit Signatur
 - `C f(C1, ..., Cn)`
 - Typischer Aufruf ist ein **Ausdruck** (expression):
 - `x = x.f(x1, ..., xn);`
- Anweisung**:
 - `void f(C1, ..., Cn)`
 - Typischer Aufruf ist eine **Anweisung** (statement):
 - `x.f(x1, ..., xn);`





Veränderbare Typen - Ergebnistypen

Veränderbare Typen (engl.: mutable Types)

- Methoden verändern ihr Objekt
 - Beispiel: Konto, Telefonbuch, Arrays

```
Konto meins = new Konto("HP",1000);
Konto deins = new Konto("Du",200);
deins.überweise(meins,500);
System.out.println(meins.getKontoStand());
```

Ergebnistypen

- Können nicht verändert werden
- Methoden liefern immer ein **neues** Objekt. Altes Objekt bleibt unverändert

- Beispiel: String, int,

```
public static void stringTest() {
    String name = "Otto";
    String langName;
    langName = name.concat("kar");
    System.out.println("langName="+langName);
    System.out.println("name="+name);
}
```

Options
langName=Ottokar
name=Otto

Der alte *name* unverändert



StringBuffer: Mutable Strings

- Ein StringBuffer ist eine veränderbare (mutable) Variante von String

- Umwandlungen mit Konstruktoren

- `StringBuffer`(String s)
- `String`(StringBuffer b)

- Methoden verändern ihr Objekt

- `StringBuffer` **append**(StringBuffer sb);
- `StringBuffer` **insert**(int offset, `String` str)

```
public static void sbTest() {
    StringBuffer name = new StringBuffer("Otto");
    StringBuffer langName;
    langName = name.append("kar");
    System.out.println("langName="+langName);
    System.out.println("name="+name);
}
```

Options
langName=Ottokar
name=Ottokar

Der alte *name* ist verändert



Veränderbare - und Ergebnistypen

- Veränderbare Typen erkennt man **off** - nicht immer - an der Signatur der Operationen:
- C sei zu implementierende Klasse,
 - C_1, \dots, C_n seien bereits vorhandene Klassen.
- Implementiere:
 - $f : C \times C_1 \dots C_n \rightarrow C$
- In einem **veränderbaren Datentyp** C typisch:
 - $\text{void } f(C_1, x_1, \dots, C_n, x_n)$
 - Aufrufe sind **Anweisungen, sie verändern ein vorhandenes Objekt**
 - $x.f(a_1, \dots, a_n)$
 - **x** wird modifiziert
- In einem **Ergebnistyp** C (engl.: immutable) als:
 - $C f(C_1, x_1, \dots, C_n, x_n)$
 - Aufrufe sind **Expressions, sie liefern ein komplett neues Objekt**
 - $y = x.f(a_1, \dots, a_n)$
 - das alte **x** bleibt bestehen

Komplexe Zahlen	
«Konstruktoren»	veränderbar
...	
«Mutatoren»	
+ void add(Complex)	
+ void mult(Complex)	
«Selektoren»	
...	
Komplexe Zahlen	
«Konstruktoren»	Ergebnistyp
...	
«Operationen»	
+ Complex plus(Complex)	
+ Complex mal(Complex)	
«Selektoren»	
....	



Behälter-Datentypen

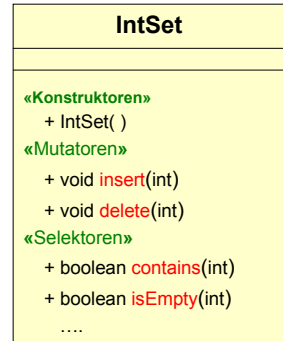
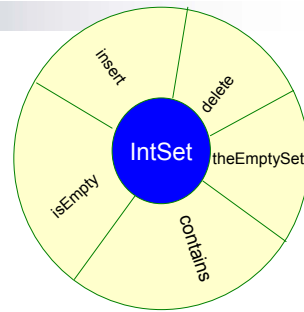
- Behälter-Datentypen dienen dazu, gleichartige Daten
 - zu speichern
 - zu finden
 - zu besuchen
- Bisher bekannte Behälter Datentyp
 - Array
 - Dateien
 - String, StringBuffer
- Andere bekannte Behälter
 - Mengen
 - Listen
 - Folgen
 - Haufen
 - Stapel
 - Schlangen
- In welchen Situationen und warum arrangiert man Dinge des täglichen Lebens
 - in einer Liste
 - auf einem Haufen
 - auf einem Stapel
 - in einer Schlange
 - In einer Menge
- Welche Vorteile bieten diese Behälter ?





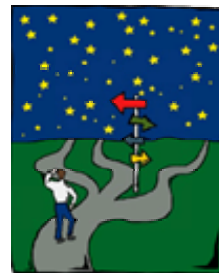
Datentyp IntSet

- Menge von Integern
- Sort: `IntSet`
- Operationen
 - `insert: IntSet × int → IntSet`
 - `delete: IntSet × int → IntSet`
 - `theEmptySet: → IntSet`
 - `contains: IntSet × int → boolean`
 - `isEmpty: IntSet → boolean`
- Gemeint sind die folgenden Operationen:
 - `theEmptySet = ∅`
 - `insert(S,i) = S ∪ {i}`
 - `delete(S,i) = S \ {i}`
 - `contains(S,i) = i ∈ S`
 - `isEmpty = (S = ∅)`



Entscheidungen

- Klasse `IntSet`
- Methoden als Objektmethoden
 - **Beispiel:**
 - `insert: IntSet × int → IntSet`
 - Zwei Möglichkeiten
 - `insert` liefert **neue** Menge
 - `IntSet insert(int i)`
 - `insert` **modifiziert** Ausgangsmenge `S`
 - `void insert(int i)`
- Wir begrenzen die Größe der Menge
 - Ist die Menge voll, so liefert `insert` eine Exception





Implementierung

- `füllStand` zeigt immer auf die nächste freie Position
- Repräsentierte Menge = { `s[i] | 0 ≤ i < füllStand` }
- Elemente `s[j]` mit `j ≥ füllStand` gelten als nicht vorhanden
- `theEmptySet` ist der Konstruktor `public IntSet`
- Falls Container voll ist, und ein weiteres Element eingefügt werden soll, wird eine Ausnahme erzeugt
- Löschen: Element wird durch `s[füllStand-1]` überschrieben

```
public class IntSet{
// Private Felder
private int maxSize=100;
private int[] s;
private int füllStand=0;

/** Konstruktor */
public IntSet(){
s = new int[maxSize];
füllStand=0;
}

/** Element einfügen */
public void insert(int i)
throws ContainerException{
if(!this.contains(i)){
if (füllStand==maxSize)
throw new ContainerException("Full");
else {
s[füllStand]=i;
füllStand++;
}
}
}

/* Element löschen */
public void delete(int i){
int index = search(i);
if (index > -1){
füllStand--;
s[index] = s[füllStand];
}
}
}
```

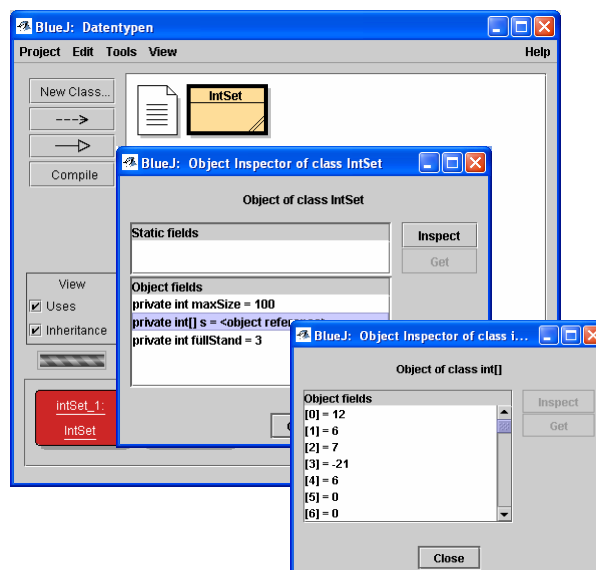
Prakt. Informatik II

© H. Peter Gumm, Philipps-Universität Marburg



Testen in Bluej

- IntSet Objekt erzeugen
 - Rechte Maustaste Klasse
- ObjektMethode aufrufen
 - Rechte Maustaste Objekt
 - Mehrere insert, delete
- Inspizieren
 - Rechte Maustaste Objekt
- Arrayfelder ansehen
 - Object reference doppelklicken
- Wieviele Elemente sind in der gezeigten Menge ?
 - 3 ?
 - 5 ?
 - 7 ?



Prakt. Informatik II

© H. Peter Gumm, Philipps-Universität Marburg



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
- Wir legen die maximale Größe durch den Konstruktor fest
- Bei der Hilfsfunktion search sollte „==“ durch „equals(...)“ ersetzt werden
- Ordnung nur möglich für Klassen, die Comparable implementieren

```
public class Set {
    // Private Felder
    private Object[] theSet;
    private int fullStand=0;

    /** Konstruktor */
    public Set(int maxSize) {
        theSet = new Object[maxSize];
        fullStand=0;
    }

    /** Ist Element schon vorhanden */
    public int search(Object o) {
        for(int k=0;k<fullStand;k++)
            if(theSet[k].equals(o)) return k;
        return -1;
    }
}
```



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 $\frac{1}{4}$ 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++;  
        }  
    }  
}
```

Prakt. Informatik II

© H. Peter Gumm, Philipps-Universität Marburg



Sets in Java

- In java.util existiert
 - Interface Set mit
 - SubInterface SortedSet
- Implementierungen
 - HashSet
 - TreeSet

The screenshot shows the Java API documentation for the `Set` interface in the `java.util` package. The page title is "Interface Set". It lists the following information:

- All Superinterfaces:** [Collection](#)
- All Known Subinterfaces:** [SortedSet](#)
- All Known Implementing Classes:** [AbstractSet](#), [HashSet](#), [LinkedHashSet](#), [TreeSet](#)

The interface definition is shown as:

```
public interface Set  
    extends Collection
```

A collection that contains no duplicate elements.

Prakt. Informatik II

© H. Peter Gumm, Philipps-Universität Marburg



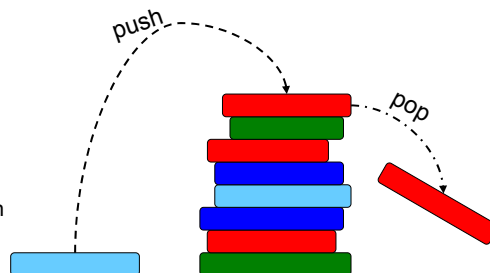
Behälter mit Zugriffsreihenfolge

- Auf die Objekte eines Behälters kann man nach anderen Kriterien zugreifen
 - Ein Stapel von Briefen
 - Man greift immer nur den obersten
 - Den zuletzt abgelegten
 - Ein Schlange von Personen an der Kasse
 - Der vorderste kommt zuerst dran
 - Der am längsten in der Schlange steht
 - Eine Schlange am Flugschalter
 - Der vorderste kommt zuerst dran
 - Die Flugzeugbesatzung darf sich jederzeit vordrängeln
 - Sie hat Priorität



Stack - Stapel

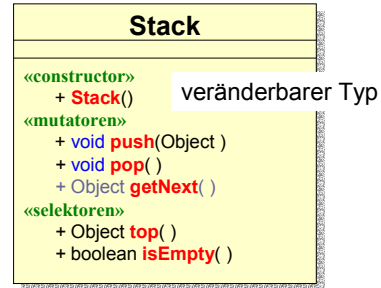
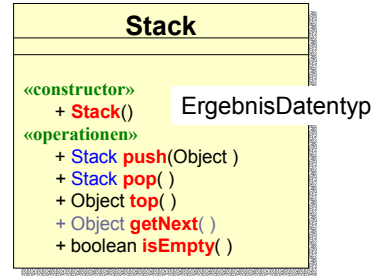
- Behälter
- direkter Zugriff nur auf letztes eingefügtes Element
 - Last in – First Out (LIFO)
 - Beispiele:
 - Stapel von Tabletts in der Mensa
 - Stapel von Klausuren
- Manche Leute sagen: „Keller“
 - Die zuletzt eingelagerten Kartoffeln werden zuerst gegessen
- Stacks sind wichtige Datenstrukturen der Informatik
 - Statt **insert** und **delete** sagt man
 - **push** und **pop**





Stack - Spezifikation

- LIFO-Behälter von Objekten
- Sort – Stack S von Objekten
- Operationen:
 - $push : Stack \times Object \rightarrow Stack$
 - $pop : Stack \rightarrow Stack$
 - $emptyStack : \rightarrow Stack$
 - $top : Stack \rightarrow Object$
 - $isEmpty : Stack \rightarrow boolean$
- Gleichungen
 - $top(push(s,e)) = e$
 - $pop(push(s,e)) = s$
 - $isEmpty(emptyStack) = true$
 - $isEmpty(push(s,e)) = false$

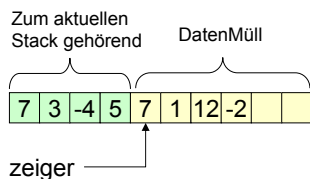


Object getNext() { Object o = top(); pop(); return o; }



Implementierung: Bounded Stack

- Wir implementieren einen Stack als bounded stack
 - d.h. mit einer begrenzten Kapazität
 - Diese wird bei dem Aufruf des Constructors bestimmt
- Wir wählen die destruktive Variante der Stackoperationen
 - void push(Element e)
 - void pop()



```
public class BoundedStack{
    // Instanz variablen           // +---+---+---+
    private Object [] theStack;    // | 5 | 7 | 3 |
    // Zeiger auf naechsten freien Platz: // +---+ ^ +---+
    private int zeiger;           // ____|

    /** Konstruktor erzeugt leeren stack */
    BoundedStack(int kapazität){
        theStack = new Object[kapazität];
        zeiger = 0;
    }

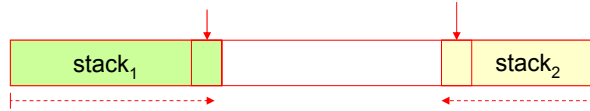
    /** push legt Objekt auf dem Stack ab. */
    void push(Object o) throws Exception{
        if (isFull())
            throw new Exception("Can't push on full stack");
        else{
            theStack[zeiger]=o;
            zeiger++;
        }
    }

    /** pop entfernt oberstes Element */
    void pop() throws Exception{
        if (isEmpty()) throw new Exception("Can't pop empty Stack");
        else zeiger--; // es muss nichts gelöscht werden
    }

    /** isEmpty() prüft, ob der stack leer ist: */
    public boolean isEmpty(){
        return zeiger == 0;
    }
}
```



ZweiStacks



- Zwei Stacks kann man in einem Array unterbringen
 - einer wächst von links nach rechts
 - der andere von rechts nach links
 - Überlauf erst, wenn Summe der Längen > Länge des Arrays
- Optimale Ausnutzung des vorhandenen Platzes
- Viele Programmiersprachen
 - z.B. Pascal
 verwalten so
 - den **heap**
 - dynamischer Speicher
 - den **runtime-stack**
 - für Methodenaufrufe
 - lokale Variablen

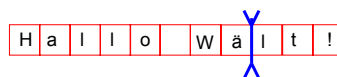


Prakt. Informatik II

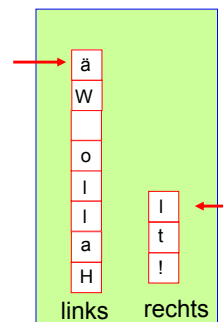
© H. Peter Gumm, Philipps-Universität Marburg



Anwendung: Text Editor



- Ein Texteditor besteht aus **Cursor**
 - einer Folge von Zeichen
 - einem Cursor
- Operationen
 - zur Bewegung des Cursors
 - zum Tippen oder Löschen
- Zur Implementierung eignet sich ein Stackpaar (links,rechts)
 - **links** : die Zeichen vor dem Cursor
 - **rechts** : die Zeichen rechts vom Cursor
- Positionierung und Operationen
 - **left()** = right.push(left.getNext())
 - **right()** = left.push(right.getNext());
 - **delete()** = left.pop()
 - **type(c)** = left.push(c)



Prakt. Informatik II

© H. Peter Gumm, Philipps-Universität Marburg



Anwendung von Stacks

- Auf dem Stack abgelegte Elemente entnimmt man in umgekehrter Reihenfolge
- WriteBinary nutzt dies aus:
 - die Binärziffern werden in falscher Reihenfolge produziert und auf dem Stack abgelegt
 - dann werden sie vom Stack entnommen und ausgegeben

```
void WriteBinary(int n) throws Exception{
    BoundedStack s = new BoundedStack(64);
    while(n > 0){
        s.push(new Integer(n % 2));
        n = n/2;
    }
    while(!s.isEmpty()){
        System.out.print(s.top());
        s.pop();
    }
}
```

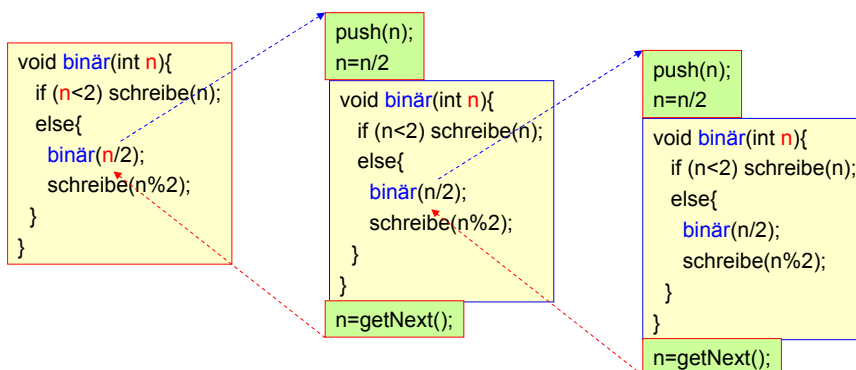
BlueJ: Method Call
void WriteBinary(int n)
stackApp_1.WriteBinary(2003)

BlueJ: Terminal Window
Options
11111010011



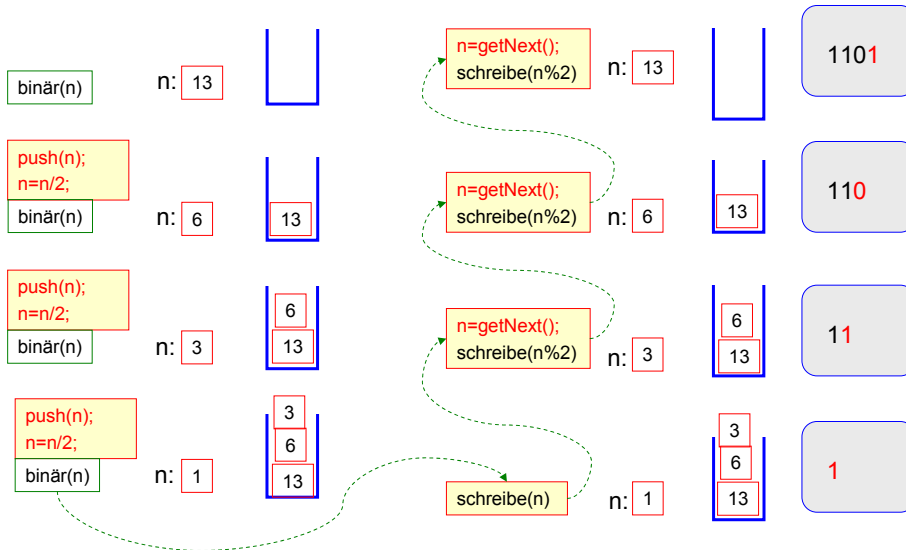
Auswertung rekursiver Methoden

- Beim Eintritt
 - Sichere alle lokalen Variablen und Parameter auf dem Stack
- Nach Beendigung
 - Lade sie wieder vom Stack





Auswertung von binär(13)



Entrekursivierung

Eine end-rekursive Funktion

$$f(x) = \begin{cases} g(x), & \text{falls } p(x) \\ h(x, f(r(x))), & \text{sonst} \end{cases}$$

Programmierung in Java

```
RTyp f(ATyp x) {
    if ( p(x) ) return g(x);
    else return h( x, f(r(x)) );
}
```

Iterative Version mit Stack s

```
RTyp f(ATyp x){
    RTyp result; ATyp arg; Stack s;
    s=new Stack();

    while( ! p(x) ) {
        s.push(x);
        x = r(x);
    }

    result = g(x);

    while (!s.isEmpty()){
        arg = (ATyp)s.getNext();
        result = h( arg, result);
    }

    return result;
}
```



Arithmetische Ausdrücke

- Für die Angabe komplexer arithmetische Ausdrücke benutzt man
 - Präzedenzen
 - Klammern
 - Beispiele:
 - $(x+1)^x + 1/x * e^{-(x+1)} * \sin(1/x)$
- Dies macht die technische Auswertung kompliziert
 - Wie berechnen Sie solche Ausdrücke mit dem Taschenrechner?
 - $2^{x(x+x^2)}, (x+1)^2/(x^2+1), \dots$
 - Geben Sie einen Algorithmus an, wie man solche Ausdrücke auswertet



Notation von Ausdrücken

- Infixnotation
 - Operationszeichen zwischen den Operanden
 - $x + 3, 2^{15}, x$ and y
 - erfordert Klammern
 - $(x+1)^{15}, x$ and $(y \text{ or } z), x / \sqrt{(x+1)}, \sin(x+1)$
- Praefixnotation
 - Operationszeichen vorne
 - $+ x 3, * 2 15,$ and $x y$
 - erfordert keine Klammern (sofern Stelligkeit der Operatoren bekannt ist)
 - $* + x 1 15,$ and $x \text{ or } y z, / x \sqrt{+ x 1}, \sin + x 1$
 - aber Operator muss auf Berechnung der Argument warten
 - wird in der Sprache LISP verwendet.
- Postfixnotation (auch UPN-umgekehrte polnische Notation)
 - Operationszeichen hinten
 - $x 3 +, 2 15 *, x y$ and
 - erfordert keine Klammern
 - $x 1 + 15 *, x y z$ or and, $x x 1 + \sqrt{/}, x 1 + \sin$
 - Operatoren haben gleich ihre fertigen Argumente



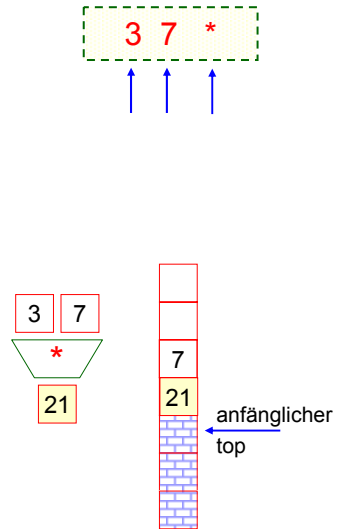
HP 35 der erste erschwingliche programmierbare Taschenrechner

Ein Beispielprogramm z.B. bei www.hpmuseum.org/software/25simeq.htm



Expression-Auswertung mit Stack

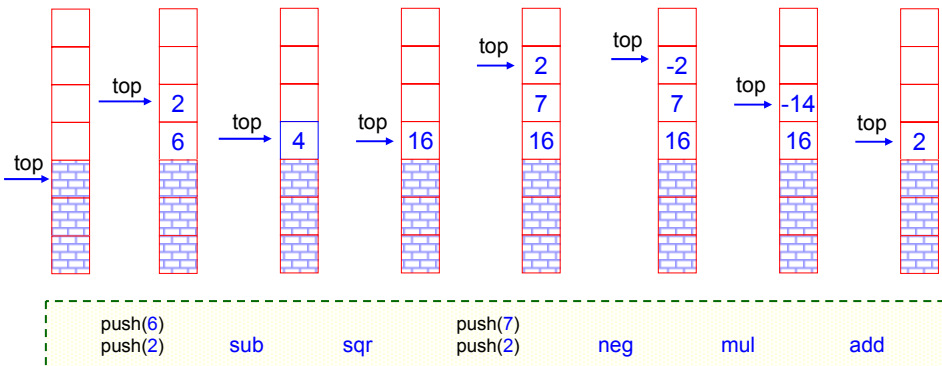
- Um eine Operation mit k-Argumenten auszuwerten
 - Lege k Argumente auf den Stack
 - push
 - Operation entnimmt oberste k Elemente
 - getNext
 - Berechnet das Ergebnis
 - Legt das Ergebnis auf dem Stack ab.
- Netto- Veränderung des Stacks:
 - Wie zu Beginn, aber das Ergebnis ist hinzugekommen und liegt obenauf



6 2 - 2 7 2 (±) * +

Postfix Ausdruck : 6 2 - 2 7 2 (±) * +

Stack-Maschinen Code: push(6) push(2) sub sqr push(7) push(2) neg mul add

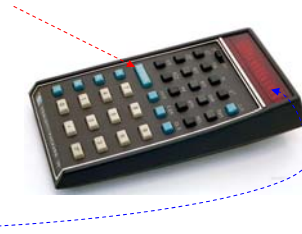


Netto: push(Ergebnis des Ausdrucks)



Auswertung von UPN mit Stack

- Zahlenwerte werden auf den Stack gelegt (push)
 - HP verwendete die **ENTER** Taste
- Operatoren (+, -, *, sin, x^y, ...)
 - holen ihre Argumente vom Stack
 - pop**
 - berechnen Ergebnis,
 - speichern das Ergebnis auf dem Stack
 - push**
 - Display zeigt immer **top** des Stacks



Infix : $1 / \sin (\ln(17) + 5 * \sqrt{3})$
 Präfix : $(1/x)(\sin(+(\ln(17), * (5, \sqrt{(3)))))$
 Postfix: $17 \ln 3 \sqrt{5} * + \sin 1/x$

HP-35 :



Stack Evaluierung: $17 \ln 3 \sqrt{5} * + \sin 1/x$

- push(17)
 - 17 auf den Stack legen

- In
 - Einstellige Operation
 - Argument: top(), entfernen mit pop()
 - Resultat berechnen und auf Stack legen

- Push(3)

- $\sqrt{\quad}$
 - Analog zu ln

- Push(5)

- \times
 - Argumente: Oberste zwei Stackelemente
 - Argumente entfernen pop(), pop()
 - Ergebnis der Operation auf Stack legen

- $+$
 - Analog zu \times

- sin
 - Analog zu $e^x, \sqrt{\quad}$

- $1/x$
 - Analog zu $e^x, \sqrt{\quad}, \sin$

Der Stack



FORTH – die Stacksprache

- FORTH ist eine Programmiersprache
 - FORTH besteht aus Kommandos („words“) und Operationen, die einen Stack manipulieren
 - Mit FORTH kann auch moderne Programme schreiben
 - objektorientiert
 - mit GUI
 - CGI, etc. ..
- FORTH ist sehr maschinennah
 - FORTH ist sehr effizient
 - Die Java-Virtual-Machine (JVM) ist eine FORTH-ähnliche Sprache
- 🔍 FORTH Programme sind für nicht eingeweihte schwer zu lesen
- 🎮 FORTH-Programmieren macht Spaß



```

\ Drei kleine FORTH-Programme:
\ Euclid's algorithmus
: UMOD ( u1 u2 - remainder)
  0 SWAP UM/MOD DROP ;
: GCD ( u1 u2 -- gcd )
  BEGIN ?DUP WHILE TUCK
    UMOD REPEAT ;
\ das gleiche rekursiv
: GCD-RECURSIVE ( u1 u2 - gcd )
  ?DUP IF TUCK
    UMOD RECURSE THEN ;
\ First 20 Fibonacci numbers
: fibonacci ( -- )
  0 1 20 0 DO DUP .
  TUCK + LOOP 2DROP ;

```



Eine Interaktion mit FORTH

```

Gforth 0.5.0, Copyright (C) 1995-
Gforth comes with ABSOLUTELY NO
Type 'bye' to exit
ok
23 17 39 5 66 ok
.s <5> 23 17 39 5 66 ok
. 66 ok
.s <4> 23 17 39 5 ok
dup ok
.s <5> 23 17 39 5 5 ok
drop ok
.s <4> 23 17 39 5 ok
rot ok
.s <4> 23 39 5 17 ok
+ ok
.s <3> 23 39 22 ok
- ok
.s <2> 23 17 ok
* ok
.s <1> 391 ok
2 / ok
.s <1> 195 ok
3 mod .s <1> 0 ok
. 0 ok
.s <0> ok
\ Wir berechnen 2+3*4-5 : ok
2 3 4 * + 5 - . 9 ok
bye

```

Zahlen und Werte werden auf den Stack gelegt

23 17 39 5 66

Im interaktiven Modus beantwortet FORTH korrekte Eingaben mit 'ok'

FORTH-words

- .s zeigt den Stack
- . pop das oberste Element und zeige es an
- swap vertausche oberste Elemente
- dup dupliziere oberstes Element
- drop pop
- rot rotiere oberste beiden Elemente
- + addiere obersten drei Elemente
- ersetze sie durch Ergebnis

-, *, /, mod analog

\ beginnt Kommentarzeile

Mehrere Kommandos auf einer Zeile – ok.



Programmieren in FORTH

- Zwischen ':' und ';' können neue FORTH-Worte (Programme) definiert werden
 - Das System antwortet mit 'compiled'
- Zuerst üben wir, wie die Funktion berechnet wird:
 - quadratsumme(5,6) = $5^2 + 6^2 = 61$
- Dann definieren wir ein Wort für die entsprechenden Aktionen
- Definition beginnt mit ':' und endet mit ';'.

```

Gforth 0.5.0, Copyright (C) 1995-2000 Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY;
Type 'bye' to exit

\ Auf dem Stack liegen zwei Zahlen
\ Berechne ihre Quadratsumme
ok
5 6 ok
.s <2> 5 6 ok
dup # ok
.s <2> 5 36 ok
swap dup # ok
.s <2> 36 25 ok
+ . 61 ok

\ Dafür definieren wir ein Wort
: quadratsumme
dup # swap dup # +
;
ok
\ Wir probieren es aus :
2 3 quadratsumme . 13 ok
bye
  
```



Postscript

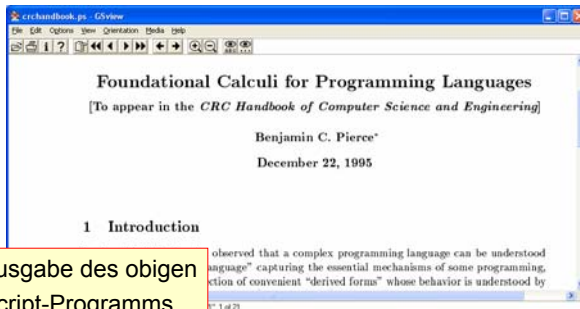
- Postscript
- Seitenbeschreibungssprache von Adobe
- Ergebnis
 - Dokumente
 - mit Graphik
 - Mit Farbe
- Beliebige Berechnungen
 - Arithmetik
 - Programme
- Theoretisch kann man damit beliebige Programme schreiben
- Postscript Dokumente sind Programme für den Postscript Interpreter
- Schauen Sie z.B. einmal mit einem Editor in eine Postscript-Datei:

Ein Postscript Dokument ist ein Programm für den Postscript Interpreter

```

%%Page: 1 1
1 0 bop 55 231 a Fs(F)-7 b(oundational)26 b(Calculi)g(for)g
(Programming) i(Languages)23 323 y Fr((T)-5 b(o)18 b(app)r(ear)h
(the)g Fq(CR)o(C) i(Handb)m(o)m(ok) i(of) f(Computer)g(Scienc)m(e)
(Engine)m(ering)p Fr(J) 694 449 y(Benjamin)17 b(C.)h(Pierce)1175
431 y Fp(\003)701 551 y Fr(Decem)n(b)r(er)e(22,)i(1995)0
812 y Fo(1)67 b(In)n(tro)r(duction)0 914 y Fn(In)17 b(the)f(mic
observ)o(ed)e(that)f(a)h(complex)h(programm
)q(e) f(understo)q(o)q(d)0 970 y(in)h(terms)f
g(language")h(capturing)f(the)h(essen)o(tial
e)g(programming,)0 1027 y(st)o(yle)e(togethe
ion)j(o) f(d)con)o(v)o(enien)o(t)h(\deriv)o(e
)q(eha)o(vior)h(is)h(understo)q(o)q(d)f(b)o(
j(them)f(in)o(to)h(the)f(core)h(\(cf.)701
  
```

Die Ausgabe des obigen Postscript-Programms





Postscript

```

GNU Ghostscript 7.05 (2002-04-22)
Copyright (C) 2002 artofcode LLC, Benic
This software comes with NO WARRANTY: s

GS>2 3 add
GS<1>=
5
GS>2 5 7 mul add 6 sub =
31
GS>/quadratsumme (
dup mul
exch dup mul
add ) def
GS>2 3 quadratsumme
GS<1>=
13
GS>100 100 moveto
GS>200 300 lineto
GS>stroke
GS>200 200 moveto
GS>0 10 300 (
200 200 moveto
100 lineto
stroke ) for
GS>/Times-Roman findfont
Loading NimbusRomNo9L-Regu font fro
20 625828 1622528 330046 0 done.
GS<1>>20 scalefont
GS<1>>setfont
GS>100 50 moveto
GS>(Hallo, Welt!) show
GS>100 300 moveto
GS>0 15 300 (
rotate
(Hallo) show ) for
GS>

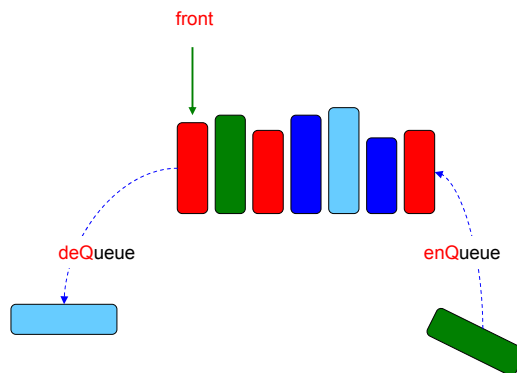
```

- Stackcode
 - Arithm. Operatoren u.a.
 - add, mul, sub, div, mod**
- Definitionen
 - beginnen mit /
 - enden mit **def**
 - Code zwischen { und }
- Stackmanipulation
 - dup, exch,**
- Graphische Operationen
 - moveto** (Bezugspunkt)
 - lineto**
 - stroke** (mache sichtbar)
- for-Schleife
 - from k to { Block } for**
- Text
 - Fontauswahl
 - scalet** Skalierung
 - Strings in runden Klammern
 - show**
 - rotate**: Textrichtung



Queues

- Behälter Datentyp
 - Zugriff immer auf Objekt, das am längsten im Behälter ist
 - First In First Out – FIFO
- Warteschlange
 - Schlange in der Bäckerei
 - An der Bushaltestelle
- Operationen
 - enqueue** – einfügen
 - dequeue** – entnehmen
 - isEmpty** – ist noch ein Element vorhanden ?
 - front** – nächstes Element





Queue - Spezifikation

- FIFO Behälter
- Sort – Queue Q von Objekten
- Operationen:
 - $enQ : Queue \times Object \rightarrow Queue$
 - $deQ : Queue \rightarrow Queue$
 - $emptyQueue : \rightarrow Queue$
 - $front : Queue \rightarrow Object$
 - $isEmpty : Queue \rightarrow boolean$
- (Bedingte) Gleichungen
 - $deQ(enQ(q,x)) = q$
 - $isEmpty(q) \Rightarrow front(enQ(q,x)) = x$
 - $\neg isEmpty(q) \Rightarrow front(enQ(q,x)) = front(q)$
 - $isEmpty(emptyQueue) = true$
 - $isEmpty(enQ(q,e)) = false$

Queue	
«Konstruktor»	+ Queue() Ergebnistyp
«operationen»	+ Queue enQ(Object) + Queue deQ() + Object front() + Object getNext()
«update»	+ boolean isEmpty()

Queue	
«Konstruktor»	+ Queue() Veränderbarer Typ
«Mutatoren»	+ void enQ(Object) + void deQ() + Object getNext()
«Selektoren»	+ Object front() + boolean isEmpty()



Beispiele von Queues

- Kanal
 - Daten müssen in der richtigen Reihenfolge ankommen
 - Sender schreibt in den Kanal,
 - Empfänger liest aus dem Kanal
- Puffer (engl.: buffer)
 - Lesen aus einer Datei
 - Festplatte schreibt in den Puffer
 - Programm liest aus dem Puffer
 - Vorteil: Zeitliche Entkopplung
 - Schreiben einer Datei
 - analog
- Cache
 - Puffer zwischen Hauptspeicher (langsam)
 - Prozessor (schnell)
- Warteschlange
 - Printerqueue
 - Programm schreibt Druckauftrag in die Printerqueue
 - Drucker arbeitet alle Druckaufträge in der Reihenfolge des Ankommens ab
- Pipe (üblich unter Unix/Linux)
 - Verbindung zweier Programme
 - Programm 1 leitet Ausgabe in pipe
 - Programm 2 entnimmt Eingabe aus der Pipe





Producer - Consumer

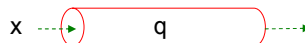
- **Produzent**
 - produziert Daten
- **Konsument**
 - Nimmt Daten entgegen
- Entkopplung durch Queue = Puffer
 - **Produzent**: enQ
 - **Konsument**: deQ
 - Vorteil: Produzent und Konsument können **asynchron** arbeiten
 - Keiner verlangsamt den anderen
- **Konkret**
 - Beim Lesen von der Festplatte:
 - **Producer**: Festplatte
 - **Consumer**: Programm
 - Bei einer Pipe
 - Programm1 ist **producer**
 - Programm2 ist **consumer**
 - Beim Senden von Daten
 - **Producer**: Sender
 - **Consumer**: Empfänger



Producer-Consumer-Protokoll

- **Producer**
 - **produziert**
 - wartet ggf. bis Queue nicht voll
 - **busy waiting**
 - fügt Element in Queue
- **Consumer**
 - wartet ggf. bis q nicht leer
 - **busy waiting**
 - entnimmt Element
 - **konsumiert es**

```
void producer(){
while(!feierabend){
produce(x);
while(q.isFull()) { }
q.enQ(x);
}
}
```



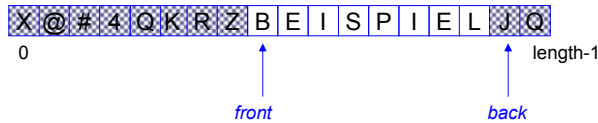
```
void consumer(){
while(!feierabend()) {
while(q.isEmpty()){}
x = q.getNext();
consume(x);
}
}
```



Queue Implementierung mit Array

- Behälter: Ein Array *theQueue*
 - Zeiger: *front*, *back*
 - *front* zeigt auf erstes Objekt
 - *back* auf den nächsten freien Platz
 - *enQ*(Object e)
 - `theQueue[back] = e; back++;`
 - *deQ*()
 - `if (!isEmpty()) front++;`
 - *isEmpty*()
 - `front == back`
- Problem
 - Bereich zwischen *front* und *back* wandert durch den Array
 - Array kann verlassen werden, auch wenn nur wenige Elemente gespeichert sind

Nur wenige Elemente in der Queue, aber schon gefährlich nahe am Abgrund



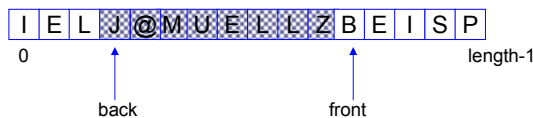
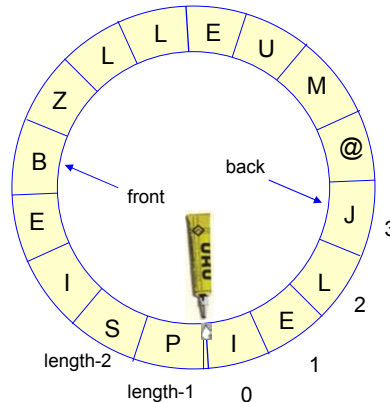
Prakt. Informatik II

© H. Peter Gumm, Philipps-Universität Marburg



Zirkuläre Arrays

- Gedanklich: Verklebe Ende des Arrays mit seinem Anfang
- Mathematisch Berechne Array-Indizes *modulo* seiner Länge
- *statt*
 - `front++`
- *rechne*
 - `front = (front+1)%length`
- *Aber wann ist der Array leer*
 - `front == back`
 - kann bedeuten
 - voll
 - leer
 - Zusätzliche Boolesche Variable
 - *empty*
 - Wird von *enQ/deQ* ggf. gesetzt



empty: `false`

Prakt. Informatik II

© H. Peter Gumm, Philipps-Universität Marburg



Implementierung der Queue

- Array mit Zeigern ...
 - **front** : erstes Element
 - **back**: Position für das nächste zu speichernde Element
- ... und
 - boolesche Variable : **full**
- Klasseninvariante ...
 - $front == back$
 ⇔ leer ⊕ voll
 - $length \geq 0$
- ... diese muss von
 - jedem Konstruktor
 - jeder Operation erhalten werden.

```
public class BoundedQueue{
// Der Behälter
private Object[] theQueue;
private int maxSize=100;

// Das vorderste Element:
private int front=0;
// Die Position für das nächste Element:
private int back=0;

// Falls front==back ist Queue voll oder leer
private boolean full=false;

// Eine Klassen-Invariante
private boolean invariante(){
return
((front==back) == (isEmpty() ^ isFull()))
&& length() >= 0 ;
}

/** Konstruktor für Queue mit Kapazität size */
public BoundedQueue(int size){
maxSize=size;
theQueue = new Object[maxSize];
assert invariante();
}
}
```



enQ, deQ, getNext

- next()
 - „biegt“ den Array zu einem Ring
 - natürlich nur, falls wir exklusiv damit im Array navigieren
- enQ()
 - prüft **full**
 - setzt es evtl.
- deQ()
 - setzt: **full=false**
- length()
 - benötigt die mathematisch korrekte Version von „modulo“

In der Mathematik gilt:
 $x \bmod p = (x + p) \bmod p$
 warum nicht in Java ???

```
private int next(int n){
return (n+1)%maxSize;
}

public void enQ(Object o) throws Exception{
if (full) throw new Exception("Queue Full");
else{
theQueue[back] = o;
back = next(back);
full = (front == back);
}
assert invariante();
}

public void deQ() throws Exception{
if (isEmpty()) throw new Exception("Queue Empty");
else{
front = next(front);
full = false;
}
assert invariante();
}

public boolean isEmpty(){
return !full && front==back;
}

public int length(){
return full? maxSize : ((back-front+maxSize) % maxSize);
}
}
```



Anwendung von Queues

- Eine einfache Simulation
 - Ein Laden hat k Kassens
 - Zu zufälligen Zeiten kommen Kunden
 - im Schnitt n pro Stunde
 - aber mal mehr – mal weniger
 - Sie laden ihre Einkaufswagen
 - mit 1 – 45 Artikeln
 - zufällig verteilt
 - gehen dann zur Kasse mit der kürzesten Schlange
 - Kassiererin braucht
 - 1 sec pro Artikel
 - 9 sec zum Kassieren
 - Wieviele Kassens müssen besetzt sein, damit
 - 90 % der Kunden
 - nicht länger als 5 min warten müssen ?

The screenshot shows the BlueJ Simulation environment. The main window displays a class hierarchy with 'Kunde' and 'BoundedQueue' as subclasses of 'Supermarkt'. Three dialog boxes are open: 'BlueJ: Create Object' for creating a 'Supermarkt' instance with 3 cashiers and 250 customers per hour, and 'BlueJ: Method Call' for calling the 'simulate' method on the 'supermar_1' object with an argument of 2*3600.

Prakt. Informatik II

© H. Peter Gumm, Philipps-Universität Marburg



Codefragment – und Ausgabe

The screenshot displays a Java code fragment for a simulation and its terminal output. The code defines a 'simulate' method that runs a loop until a specified time. It generates random customer arrivals, assigns them to the shortest queue, and simulates their service. The terminal output shows the simulation results, including customer arrival times, service times, and queue lengths.

```
public void simulate(int n){
    while( zeit < n){
        if ( Math.random()*3600 < kundenProStunde ){
            // kommt ein Kunde
            kundenzaehler++;
            int wasKauftEr = (int)(Math.random()*(maxArt-minArt) + minArt);
            Kunde kunde = new Kunde(kundenzaehler,zeit,wasKauftEr);
            System.out.print(uhrZeit()+" : Neuer Kunde"+kunde.kdNr
                +" mit " + kunde.noOfItems + " Artikeln ");

            // schicke ihn an die kurzeste Schlange
            int i = kuerzesteSchlange();
            try { kassen[i].enQ(kunde);
                System.out.println(" an Kasse |"+i);
            }catch (Exception e) {};
        }
        // Jede Kassiererin arbeitet eine Sekunde
        for(int k=0; k < anzahlKassen; k++){
            kassieren(kassen[k]);
        }
        zeit++;
    }
}
```

Terminal Output:

```
Options
01:57.38 Uhr Kunde 496 fertig. Wartezeit : 19
01:57.55 Uhr : Neuer Kunde499 mit 60 Artikeln an Kasse 1
01:57.59 Uhr : Neuer Kunde500 mit 56 Artikeln an Kasse 1
01:57.59 Uhr Kunde 495 fertig. Wartezeit : 73
01:58.16 Uhr Kunde 498 fertig. Wartezeit : 40
01:58.18 Uhr : Neuer Kunde501 mit 47 Artikeln an Kasse 0
01:58.23 Uhr Kunde 497 fertig. Wartezeit : 62
01:58.38 Uhr : Neuer Kunde502 mit 35 Artikeln an Kasse 2
01:58.41 Uhr : Neuer Kunde503 mit 39 Artikeln an Kasse 0
01:58.55 Uhr Kunde 499 fertig. Wartezeit : 60
01:59.00 Uhr : Neuer Kunde504 mit 34 Artikeln an Kasse 1
01:59.05 Uhr Kunde 501 fertig. Wartezeit : 47
01:59.13 Uhr Kunde 502 fertig. Wartezeit : 35
01:59.15 Uhr : Neuer Kunde505 mit 52 Artikeln an Kasse 2
01:59.21 Uhr : Neuer Kunde506 mit 14 Artikeln an Kasse 0
01:59.45 Uhr Kunde 503 fertig. Wartezeit : 64
01:59.52 Uhr Kunde 500 fertig. Wartezeit : 113
Anzahl der Kunden 506
Maximale Wartezeit 241
Maximale Länge einer Schlange 6
```

Prakt. Informatik II



DeQue

■ Double ended Queue

- kombiniert Stack und Queue
 - insertAtFront = push
 - removeAtFront = pop = deQ
 - first = top = front
 - insertAtRear = enQ

- Zusätzlich
 - removeAtRear
 - last

- Implementierung

