Rekursion

Was heißt "rekursiv"

Eine Methode *m()* heißt rekursiv, wenn sie sich selbst aufruft

```
m() {  m();          }      direkt rekursiv
m() {  n()  { m();   } }    indirekt rekursiv
```

Beispiel: Berechnung der Fakultät (n!)

$$n! = \underbrace{1 * 2 * 3 * \ldots * (n-1)}_{(n-1)!} * n$$

rekursive Definition

$n! = (n-1)! * n$

$1! = 1$

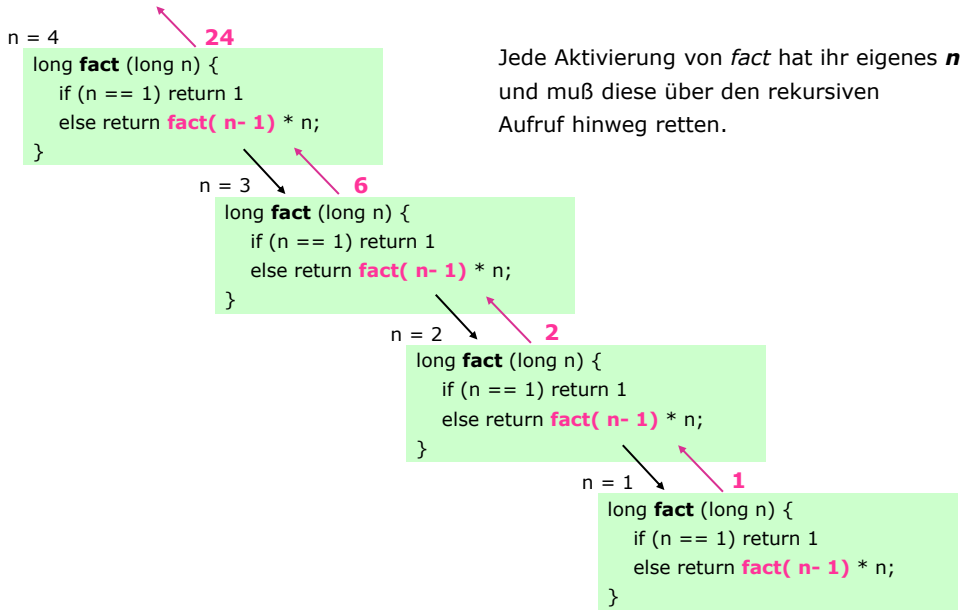
Rekursive Methode zur Berechnung der Fakultät

```
long fact (long n) {
   if (n == 1)
      return 1;
   else
   return fact( n- 1) * n;
}
```

allgemeines Muster

if (*Problem klein genug*)
  nichtrekursiver Zweig;
else
  rekursiver Zweig

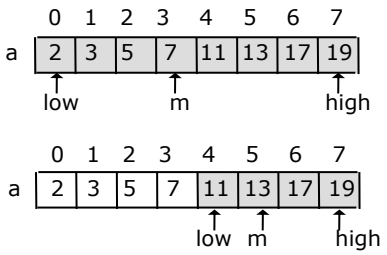
Ablauf einer rekursiven Methode



Jede Aktivierung von *fact* hat ihr eigenes ***n*** und muß diese über den rekursiven Aufruf hinweg retten.

Beispiel: binäres Suchen rekursiv

z. B. Suche von 17 (Array muß sortiert sein)



- Index *m* des mittleren Elements bestimmen
- 17 > *a*[*m*] in rechter Hälfte weitersuchen

```
static int search (int elem, int[] a, int low, int high) {
    if (low > high) return -1;          // empty
    int m = (low + high) / 2;
    if (elem == a[ m]) return m;
    if (elem < a[ m]) return search( elem, a, low, m- 1) ;
    return search( elem, a, m+ 1, high) ;
}
```

} nichtrekursiver Zweig

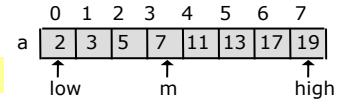
} rekursiver Zweig

Ablauf des rekursiven binären Suchens

elem = 17, low = 0, high = 7 — 6

```
static int search (int elem, int[] a, int low, int high) {
    if (low > high) return -1;
    int m = (low + high) / 2;
    if (elem == a[ m]) return m;
    if (elem < a[ m]) return search( elem, a, low, m- 1);
    return search( elem, a, m+ 1, high);
}
```

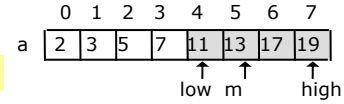
m = 3



low = 4, high = 7 — 6

```
static int search (int elem, int[] a, int low, int high) {
    if (low > high) return -1;
    int m = (low + high) / 2;
    if (elem == a[ m]) return m;
    if (elem < a[ m]) return search( elem, a, low, m- 1);
    return search( elem, a, m+ 1, high);
}
```

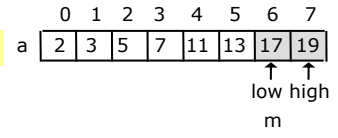
m = 5



low = 6, high = 7 — 6

```
static int search (int elem, int[] a, int low, int high) {
    if (low > high) return -1;
    int m = (low + high) / 2;
    if (elem == a[ m]) return m;
    if (elem < a[ m]) return search( elem, a, low, m- 1);
    return search( elem, a, m+ 1, high);
}
```

m = 6



Beispiel: größter gemeinsamer Teiler

rekursiv

```
static int ggt (int x, int y) {
    int rest = x % y;
    if (rest == 0) return y;
    else return ggt( y, rest) ;
}
```

iterativ

```
static int ggt (int x, int y) {
    int rest = x % y;
    while (rest != 0){
    x = y; y = rest;
    rest = x % y;
    }
    return y;
}
```

Jeder rekursive Algorithmus kann auch iterativ programmiert werden

- rekursiv: meist kürzerer Quellcode
- iterativ: meist kürzere Laufzeit

Rekursion bei rekursiven Datenstrukturen nützlich (Bäume, Graphen, ...)

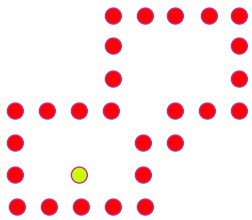
Computer Graphik: Randfüll Algorithmen

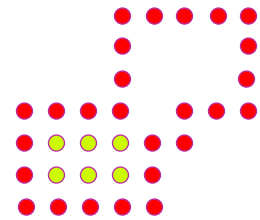
Auffüllen einer Fläche :

Bei nicht parametrisierbaren Rändern kann die Randbestimmung für einen Auffüllalgorithmus sehr aufwändig sein!

--> Auffüllen der Fläche von innen:
Starte bei beliebigem inneren Punkt und fülle die Fläche von innen nach außen!

Wie viele Nachbarpunkte sind zu beachten?





Rekursiver Randfüll Algorithmus (4, 8 Punkte)

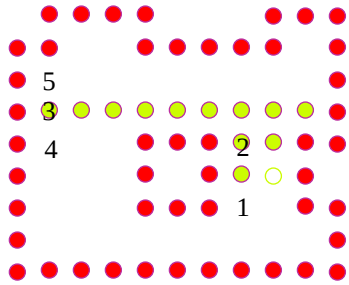
```
void boundaryfill4 ( int x, int y, int fillColor, int boundaryColor)
{
   int currentColor;

   currentColor= getpixelColor( x, y );
   if  ( ( currentColor != boundaryColor ) && (currentColor != fillColor ) ) {
      setColor ( fillColor );
      setPixel  (  x, y );
      boundaryfill4 ( x+1,    y, fillColor, boundaryColor);
      boundaryfill4 ( x -1,    y, fillColor, boundaryColor);
      boundaryfill4 ( x ,   y+1, fillColor, boundaryColor);
      boundaryfill4 ( x ,   y -1, fillColor, boundaryColor);
  }
 }
```

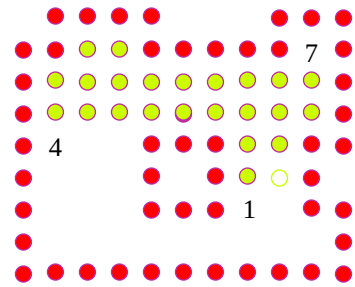
C-Code

Rekursive Randfüll Algorithmen

Nachteil: Enorme Buchhaltungskosten!

----> Durch Kombination mit Rasterlinien Algorithmen lassen sich die Buchhaltungskosten auf die Anfangspunkte neuer Rasterlinien beschränken.





Flutungs Algorithmen

Flächen, die nicht durch den Rand definiert sind, können mit Flutungsalgorithmen verändert werden.

Flutungsalgorithmen sind den Randfüllalgorithmen ähnlich.

```
void floodFill4 ( int x, int y, int fillColor, int oldColor)
{
if  ( getPixel (x,y) == oldColor) {
        setColor ( fillColor);
        setPixel  (  x, y );
        floodFill4  ( x+1,    y, fill, oldColor);
        floodFilll4 ( x -1,    y, fill, oldColor);
        floodFill4  ( x ,   y+1, fill, oldColor);
        floodFill4  ( x ,   y -1, fill, oldColor);
   }
 }
```

Schrittweise Verfeinerung

Entwurfsmethode für Algorithmen

Wie kommt man von der Aufgabenstellung zum Programm?

Beispiel

"Zähle die Häufigkeit von Wörtern in einem Text"

Welche Befehle würde man sich wünschen, um diese Aufgabe zu lösen?

- lies Wort
- speichere Wort in Tabelle
- erhöhe Wortzähler
- drucke Zähler für alle Worte

Leider gibt es keine Sprache mit diesen Befehlen

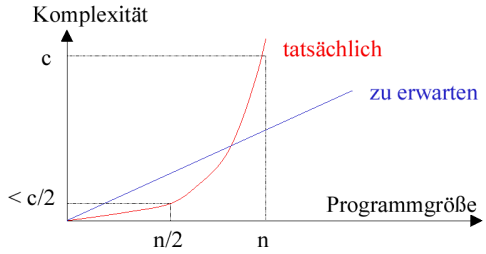
Befehle als Methoden implementieren (d. h. gewünschte "Sprache" selbst bauen).

Vorgehensweise

Schrittweise Verfeinerung

1. Zerlege Aufgabe in Teilaufgaben und spezifiziere ihre Schnittstelle
2. Nimm an, daß die Teilaufgaben schon gelöst sind
3. Implementiere Gesamtaufgabe mit Hilfe der Teillösungen
4. Sind die Teilaufgaben einfach genug?
   ja => implementiere sie direkt in einer Programmiersprache
   nein => Zerlege sie weiter (Schritt 1)

Was nützt dies?



Komplexität steigt überproportional mit der Programmgröße

Halbierung der Programmgröße reduziert die Komplexität um mehr als die Hälfte!

Beispiel

Aufgabe : "Zähle die Häufigkeit von Wörtern in einem Text"

1. Zerlege Aufgabe in Teilaufgaben und spezifiziere ihre Schnittstelle

`word = readWord();` liefert nächstes Wort oder null, wenn kein Wort mehr von der aktuellen Eingabedatei gelesen werden kann."

Klasse *WordTable* mit folgenden Methoden:

`table. count( word);` trägt word in tab ein (falls noch nicht vorhanden) und erhöht Worthäufigkeit um 1

`table. print();` gibt Wörter und ihre Häufigkeiten aus

2. Nimm an, daß die Teilaufgaben schon gelöst sind

Beispiel (Forts.)

3. Implementiere Gesamtaufgabe mit Hilfe der Teillösungen

```
class WordCount {
    public static void main (String[] arg) {
        WordTable table = new WordTable();
        In. open(" input. txt");
        String word = readWord();
        while (word != null) {
            table. count( word);
            word = readWord();
        }
        In. close();
        table. print();
    }
}
```

Beispiel (Forts.)

4. Zerlege Teilaufgaben weiter (*readWord*)

ch = In. read();	lies ein Zeichen; **In. done == false**, wenn Dateiende
Character. isLetter(ch)	prüfe, ob **ch** ein Buchstabe ist
word. append(ch)	füge **ch** an das Wort **word** an

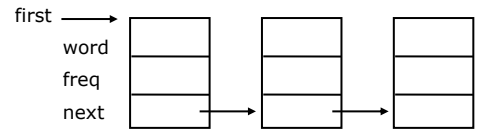
Alle Teilaufgaben sind bereits in Java implementiert.
Implementiere *readWord()* mit ihnen

```
static String readWord () {
    StringBuffer word = new StringBuffer();
    char ch;
    //----- skip nonletters
    do ch = In. read();    while ( In. done() && ! Character. isLetter( ch) );
    //----- build the word
   while ( In. done() && Character. isLetter( ch) ) {
        word. append( ch);
        ch = In. read();
    }                          // ! In. done() || ch is not a letter
   if (word. length() > 0) return    word. toString();       else return null;
 }
```

Beispiel (Forts.)

Grobstruktur der Worttabelle:

Datenstruktur:
verkettete Liste von Wörtern
und Häufigkeiten




```
class Element {
    String word;
    int freq;
    Element next;
    Element (String w) {
        word = w; freq = 1;
    }
}
```

```
class WordTable {
    Element first;
    void count (String word) {...}
    void print () {...}
}
```

Beispiel (Forts.)

4. Zerlege Teilaufgaben weiter (*count*)

elem = table. find(word);	suche *word* in *table* und liefere entspr. Element oder Null,
table. enter(word);	trage *word* in *table* ein (es kommt noch nicht vor),
freq++;	erhöhe Worthäufigkeit.

```
void count (String word) {
    Element e = table. find( word);
    if (e == null) table. enter( word); else e. freq++;
}
```

find und *enter* sind so einfach, daß man sie sofort implementieren kann.

```
Element find (String word) {
    Element e = first;
    while (e != null && !word. equals( e. word))
        e = e. next;
        // e == null || word. equals( e. word)
    return e;
}
```

```
void enter (String word) {
    Element e = new Element( word);
    e. next = first;
     first = e;
}
```

Beispiel (Forts.)

4. Zerlege Teilaufgaben weiter (print).
   Ist so einfach, daß man es sofort implementieren kann!

```
void print () {
   for (Element e = first; e != null; e = e. next)
       System.out. println( e. word + ": " + e. freq);
}
```

Zusammensetzen der einzelnen Teile

```
class Element {
    String word;
    int freq;
    Element next;
    Element (String w) {
       word = w; freq = 1;
    }
}
```

```
class WordTable {
   Element first = null;

   Element find(String word) {
        Element e = first;
        while (e != null && !word. equals( e.word)) e = e. next;
        return e;
    }

   void enter(String word) {
      Element e = new Element( word);
      e. next = first; first = e;
   }

   void count(String word) {
      Element e = find( word);
      if (e == null) enter( word); else e. freq++;
    }

   void print() {
        for (Element e = first; e != null; e = e. next)
        System.out. println( e. word + ": " + e. freq);
  }
}
```

Zusammensetzen der einzelnen Teile

```
class WordCount {

   public static void main (String[] arg) {
       WordTable table = new WordTable();
       In. open(" input. txt");
       String w = readWord();
       while (w != null) {    table. count( w);   w = readWord();   }
       In. close();
       table. print();
   }

   static String readWord () {
        StringBuffer word = new StringBuffer();
         char ch;
         do ch = In. read(); while (In. done() && ! Character. isLetter( ch));
         while (In. done() && Character. isLetter( ch)) {
             word. append( ch);
             ch = In. read();
         }
         if (word. length > 0) return word. toString(); else return null;
    }
}
```

Pakete

Idee

Paket = Sammlung zusammengehöriger Klassen (Bibliothek).

Zweck

- mehr Ordnung in Programme bringen,
- bessere Kontrolle der Zugriffsrechte (wer darf auf was zugreifen),
- Vermeidung von Namenskonflikten.

Beispiele

Paket	*enthaltene Klassen*
java. lang	System, String, Integer, Character, Object, Math, ...
java. io	File, InputStream, OutputStream, Reader, Writer, ...
java. awt	Button, CheckBox, Frame, Color, Cursor, Event, ...
java. util	ArrayList, HashTable, BitSet, Stack, Vector, Random, ...
...	...

Anlegen von Paketen

Datei *Circle. java*

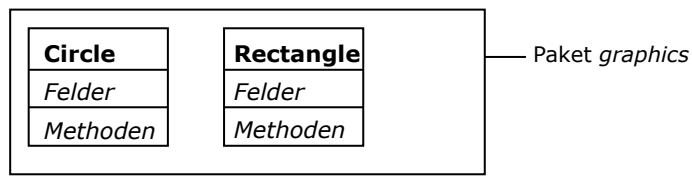
```
package graphics;

class Circle {
   ...
}
```

Datei *Rectangle. java*

```
package graphics;

class Rectangle {
   ...
}
```

← 1. Zeile der Datei

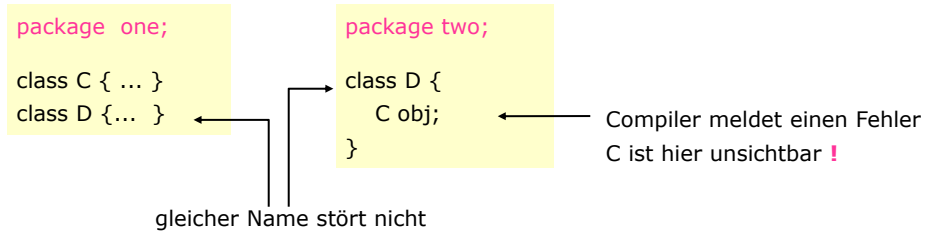
Paket *graphics* enthält die Klassen *Circle* und *Rectangle*



Wenn *package*- Zeile fehlt, gehören die Klassen zu einem namenlosen Standardpaket.

Pakete als Sichtbarkeitsgrenzen

Was in einem Paket deklariert ist, ist in anderen Paketen unsichtbar

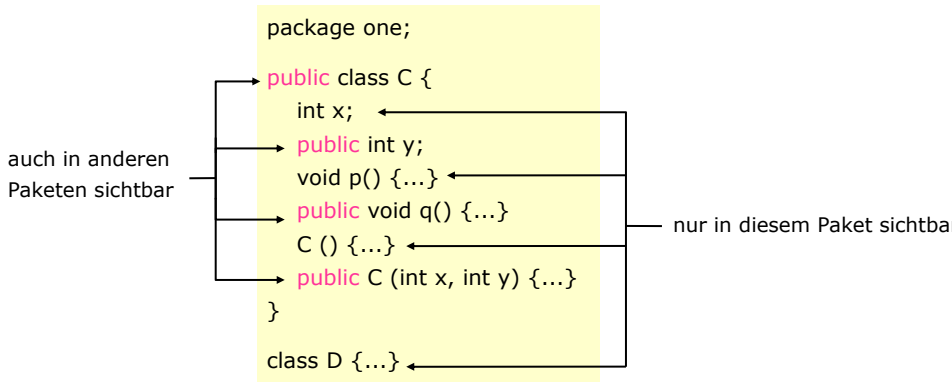


Zweck

- In verschiedenen Paketen können gleiche Namen verwendet werden.
- Schutz vor (unabsichtlicher) Zerstörung.

Pakete als Sichtbarkeitsgrenzen

Namen können mit dem Zusatz *public* exportiert werden
(sie sind dann in anderen Paketen sichtbar).



public- Felder und -Methoden werden nur dann exportiert, wenn die Klasse selbst *public* ist.

Lokale Variablen und Parameter können nicht exportiert werden.

Pakete als Sichtbarkeitsgrenzen

Exportierte Klassennamen können in anderen Paketen importiert werden.

Durch gezielten Import der Klasse

```
package myPack;

import graphics.Circle;
import one.C;
class MyClass {
   Circle c;
   ...
}
```

Durch Qualifikation mit dem Paketnamen

```
package myPack;

class MyClass {
   graphics.Circle c1;
   java.awt.Circle c2;
   ...
}
```

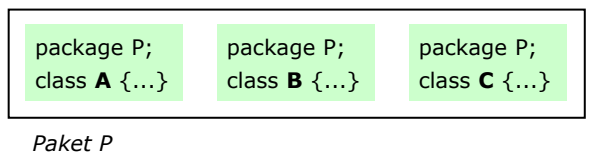
Durch Import **aller** public- Klassen eines Pakets.

```
package myPack;

import graphics.*;
class MyClass {
   Circle c;
   Rectangle r;
   ...
}
```

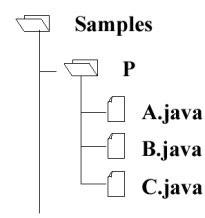
Pakete und Verzeichnisse

Pakete werden auf Verzeichnisse abgebildet, Klassen auf Dateien.

Klasse C	Datei C. java
Paket P	Verzeichnis P



Paket P



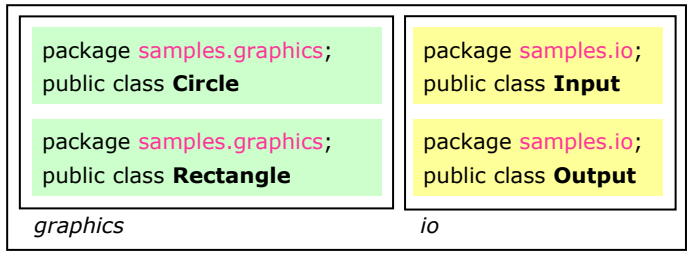
Übersetzung und Ausführung mit dem JDK.

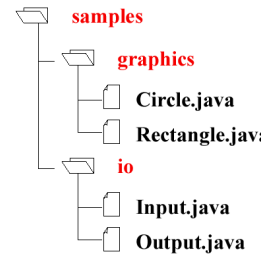
```
cd C:Samples
javac  P/A.java
java   P/A
java   P.A
```

} beides möglich

Geschachtelte Pakete

Pakete können zu größeren Paketen zusammengefaßt werden.





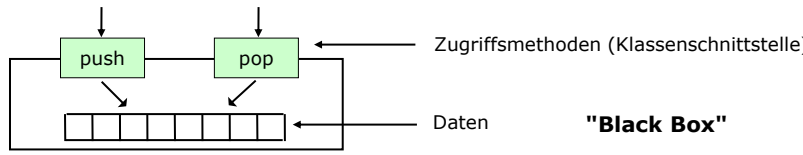
Benutzung

import samples. graphics. Circle;	importiert die Klasse Circle
import samples. graphics.*	importiert alle public- Klassen aus samples. graphics
import samples.*;	importiert alle public- Klassen aus samples (nicht aus samples. graphics)
samples.io.Output out;	Qualifikation einer Klasse aus einem geschachtelten Paket

Information Hiding (Geheimnisprinzip)

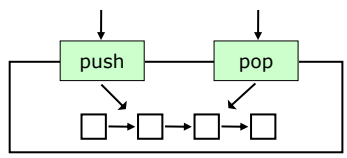
Prinzip

- Verstecke die Implementierung komplexer Datenstrukturen vor den Klienten.
- Erlaube den Zugriff auf die Daten nur über Methoden.



Warum?

- Verringert die Komplexität (Arbeiten mit den Daten wird einfacher).
- Implementierung der Daten kann geändert werden, ohne daß Klienten etwas merken.
- Schutz vor mutwilliger oder unabsichtlicher Zerstörung.

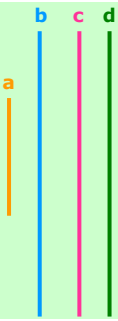


Sichtbarkeitsattribute

für Felder und Methoden

private	int a;	nur in der Klasse sichtbar, in der das Element deklariert wurde
	int b;	nur im Paket sichtbar, in dem das Element deklariert wurde
public	int c;	auch in anderen Paketen sichtbar, wenn importiert
protected	int d;	sichtbar - in der deklarierenden Klasse - in deren Unterklassen - im deklarierenden Paket

```
package one;

public class C {
    private int a;
    int b;
    public int c;
    protected int d;
}
public class D {
...
}
```

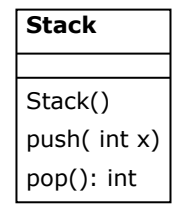



```
package two;

import one.C;
public class E extends C {
...
}
public class F {
...
}
```

Beispiel: Stack mit Information Hiding

```
public class Stack {

    private int[] data;
    private top;

    public Stack (int size) {
        data = new int[ size]; top = -1;
    }

    public void push (int x) {
        if (top >= data. length) error(" stack overflow");
        else data[++ top] = x;
    }

    public int pop () {
        if (top < 0) error(" stack underflow");
        else return data[ top--];
    }

    private void error (String msg) {
        Out. println( msg);
        System. exit( 0);
    }
}
```

Klassenschnittstelle



Dokumentationskommentare

Dokumentationskommentare

/** ... */

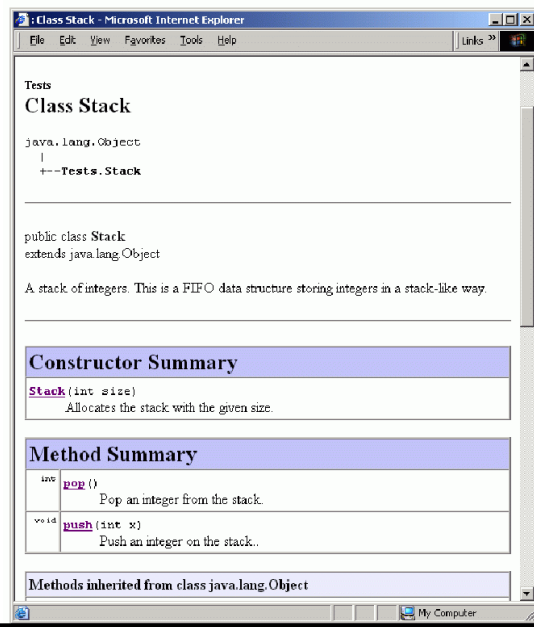
können vor die Deklarationen von Klassen, Methoden, Feldern gesetzt werden.
Werkzeug javadoc erzeugt daraus Dokumentation in HTML.

```
/** A stack of integers.
    This is a FIFO data structure storing integers in a stack- like way. */
public class Stack {
    /** The elements in the stack. */
    private int[] data;
    ...
    /** Push an integer on the stack.
        If the stack is full an error is reported and the program stops. */
    public void push (int x) {
    ...
    }
    ...
}
```

1. Satz bis Punkt wird in Kurzdoku übernommen

Rest wird in Langdoku übernommen

Erzeugte HTML- Datei (Stack. html)



javadoc

Aufrufsyntax

javadoc [options] {filename | packagename}

Optionen

- -public zeigt nur public- Deklarationen
- -private zeigt public- und private- Deklarationen
- -d path gibt Zugriffspfad für mehrere Klassen und Pakete an
- ...

Beispiel

javadoc -public myDirectory/Stack.java

erzeugt:
myDirectory/Stack.html
diverse andere Übersichtsdateien

Vollständige Dokumentation von javadoc

http:// java. sun. com/ j2se/ javadoc/

Erzeugte HTML- Datei (Stack. html)