

Mengen von Objekten und dynamische Datenstrukturen



- Menge von Objekten kann man mit Arrays folgend verwalten
 - Array von Referenzen fixer Länge; alle Stellen mit null initialisiert
 - Zähler wieviele Objekte wirklich gespeichert sind
 - Einfügen eines neuen Objekts durch Speichern an der nächsten freien Stelle

Vorteil:

- einfache Realisierung

Nachteile:

- fixe Größe des Arrays
- Speicherverschwendung bei wenigen Objekten
- Array kann voll werden; Wachsen ist schwierig
- Löschen von Elementen ist aufwendig
- Sortiertes Einfügen von Objekten ist aufwendig

Beispiel Mengen von Objekten mit Arrays

```
class Person {  
    String name;  
    int phone;  
}
```

```
class PhoneBook {  
    Person[] book = new Person[1000];  
    int nEntries = 0; // current number of entries in book  
  
    void enter (String name, int phone) {  
        if (nEntries >= book.length) Out.println("--- phone book full");  
        else {  
            book[nEntries] = new Person();  
            book[nEntries].name = name;  
            book[nEntries].phone = phone;  
            nEntries++;  
        }  
    }  
}
```

```
int lookup (String name) {  
    int i = 0;  
    while (i < nEntries && !name.equals(book[i].name)) i++;  
    // i >= nEntries || name.equals(book[i].name)  
    if (i < nEntries) return book[i].phone; else return -1;  
}
```



Beispiel Mengen von Objekten mit Arrays (2)

```
...
public static void main (String[] arg) {
    PhoneBood privBook = new PhoneBook();
    //----- read the phone book from a file
    In.open("phonebook.txt");
    String name = In.readName();
    int phone;
    while (In.done()) {
        phone = In.readInt();
        privBook.enter(name, phone);
        name = In.readName();
    }
    In.close();
    //----- search in the phone book
    for (;;) {
        Out.print("Name: "); name = In.readName();
        if (!In.done()) break;
        phone = privBook.lookup(name);
        if (phone > 0) Out.println("phone number = " + phone);
        else Out.println(name + " unknown");
    }
} // end PhoneBookSample
```



Stack (Kellerspeicher)

```
push(x);  
x = pop();
```

fügt x hinten an den Stack an
entfernt und liefert hinterstes Stackelement

```
push(3);
```



```
push(4);
```



```
x = pop();
```



```
y = pop();
```

// y == 3

LIFO-Datenstruktur
(last in first out)



Queue (Puffer, Schlange)

```
put(x);  
x = get();
```

fügt x hinten an die Queue an
entfernt und liefert vorderstes Queueelement

```
put(3);
```



```
put(4);
```



```
x = get();
```



```
y = get();
```

// y == 4

FIFO-Datenstruktur
(first in first out)



Klasse Stack

```
class Stack {  
    int[] data;  
    int top;  
  
    Stack (int size) {  
        data = new int[size];  
        top = -1;  
    }  
  
    void push (int x) {  
        if (top >= data.length-1)  
            Out.println("-- overflow");  
        else  
            data[++top] = x;  
    }  
  
    int pop () {  
        if (top < 0) {  
            Out.println("-- underflow");  
            return 0;  
        } else  
            return data[top--];  
    }  
}
```



Benutzung

```
Stack s = new Stack(10);  
s.push(3);  
s.push(6);  
int x = s.pop() + s.pop();  
// x == 9
```

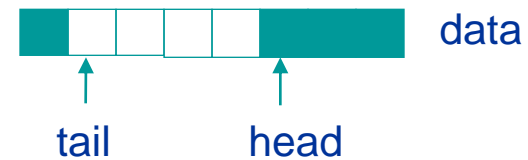
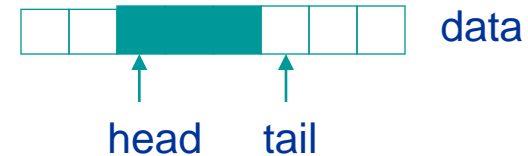
Klasse Queue

```
class Queue {
    int[] data;
    int head, tail;

    Queue (int size) {
        data = new int[size];
        head = 0;
        tail = 0;
    }

    void put (int x) {
        if ((tail+1) % data.length == head)
            Out.println("-- overflow");
        else {
            data[tail] = x;
            tail = (tail+1) % data.length;
        }
    }

    int get () {
        if (head == tail) {
            Out.println("-- underflow");
            return 0;
        } else {
            int x = data[head];
            head = (head+1) % data.length;
            return x;
        }
    }
}
```



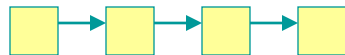
Benutzung

```
Queue q = new Queue(10);
q.put(3);
q.put(6);
int x = q.get(); // x == 3
int y = q.get(); // y == 6
```

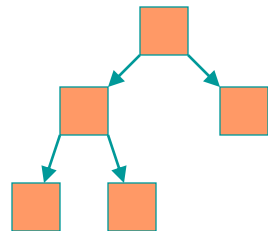


- Elemente werden zur Laufzeit (dynamisch) mit *new* angelegt
- Datenstruktur kann dynamisch wachsen und schrumpfen
- Dynamische Datenstrukturen werden aus Knoten mit Verkettungen (Referenzen) aufgebaut

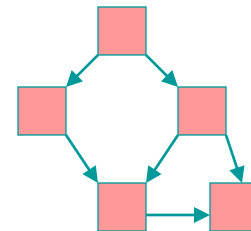
Wichtigste dynamische Datenstrukturen



Liste



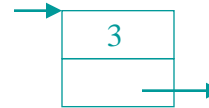
Baum



Graph

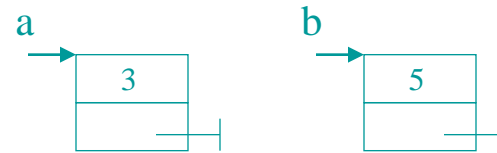
Verknüpfen von Knoten

```
class Node {  
    int val;  
    Node next;  
    Node(int v) {val = v;}  
}
```



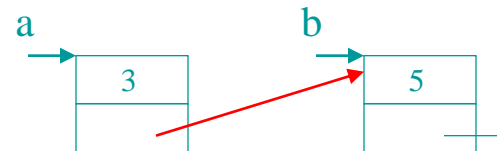
Erzeugen

```
Node a = new Node(3);  
Node b = new Node(5);
```

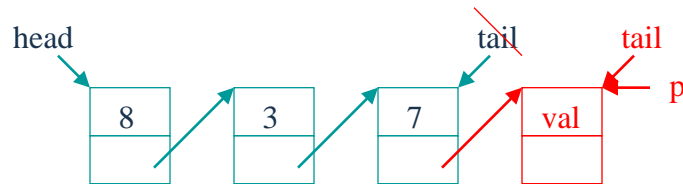


Verknüpfen

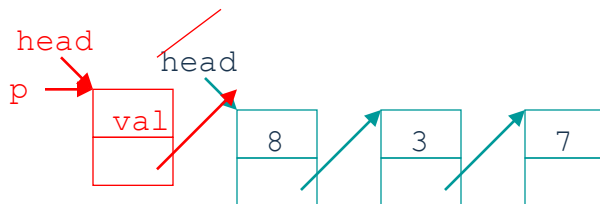
```
a.next = b;
```



Unsortierte Liste



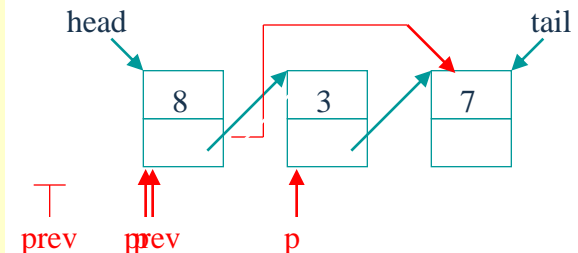
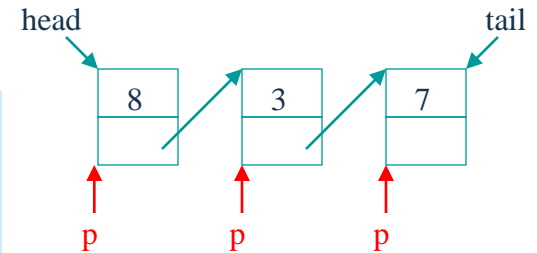
```
class List {  
    Node head = null, tail = null;  
  
    void append (int val) { // Einfügen am Listende  
        Node p = new Node(val);  
        if (head == null) head = p;  
        else tail.next = p;  
        tail = p;  
    }  
  
    void prepend (int val) { // Einfügen am Listenanfang  
        Node p = new Node(val);  
        p.next = head; head = p;  
    }  
}
```



Unsortierte Liste (Forts.)

```
class List {  
    Node head = null, tail = null;  
    ...  
    Node search (int val) { // Suchen eines werts  
        Node p = head;  
        while (p != null && p.val != val) p = p.next;  
        // p == null || p.val == val  
        return p;  
    }  
}
```

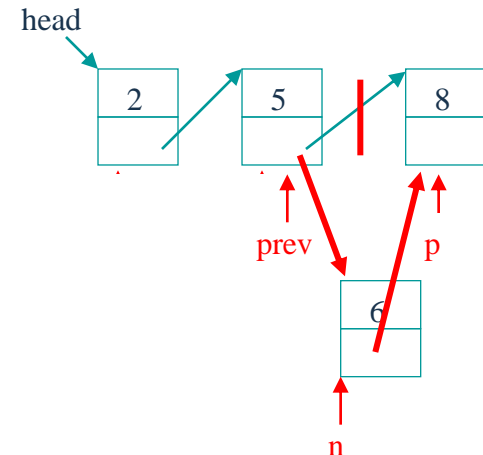
```
void delete (int val) { // Löschen eines werts  
    Node p = head, prev = null;  
    while (p != null && p.val != val) {  
        prev = p; p = p.next;  
    }  
    // p == null || p.val == val  
    if (p != null) // p.val == val  
        if (p == head) head = p.next;  
        else prev.next = p.next;  
}
```



Sortierte Liste

- Mit dynamischen Listen lassen sich durch sortiertes Einfügen einfach sortierte Mengen aufbauen

```
class List {
    Node head = null;
    ...
    void insert(int val) {
        Node p = head, prev = null;
        while (p != null && p.val < val) {
            prev = p; p = p.next;
        }
        Node n = new Node(val);
        n.next = p;
        if (p == head) {
            head = n; // insert in front
        } else {
            prev.next = n;
        }
    }
}
```



Pakete



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

java.lang
java.io
java.awt
java.util
...

enthaltene Klassen

System, String, Integer, Character, Object, Math, ...
File, InputStream, OutputStream, Reader, Writer, ...
Button, CheckBox, Frame, Color, Cursor, Event, ...
ArrayList, Hashtable, BitSet, Stack, Vector, Random, ...
...

Anlegen von Paketen

Datei *Circle.java*

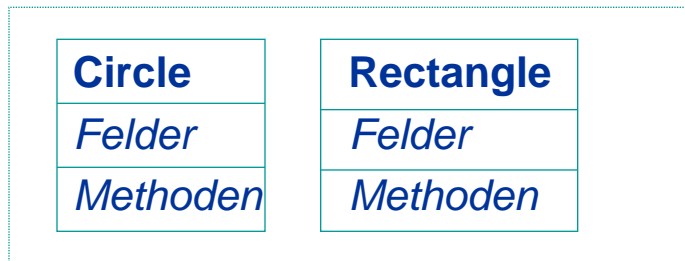
```
package graphics;  
class Circle {  
    ...  
}
```

Datei *Rectangle.java*

```
package graphics;  
class Rectangle {  
    ...  
}
```

← 1. Zeile der Datei

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



Paket *graphics*

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

Pakete werden auf Verzeichnisse abgebildet, Klassen auf Dateien

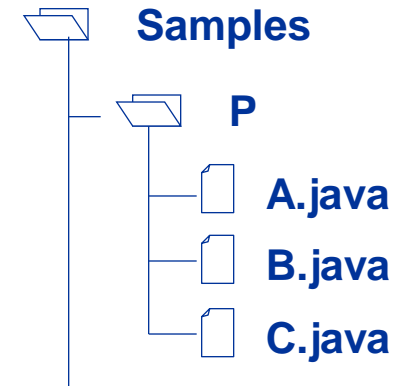
Klasse C \Rightarrow Datei C.java
Paket P \Rightarrow Verzeichnis P

```
package P;  
class A {...}
```

```
package P;  
class B {...}
```

```
package P;  
class C {...}
```

Paket P



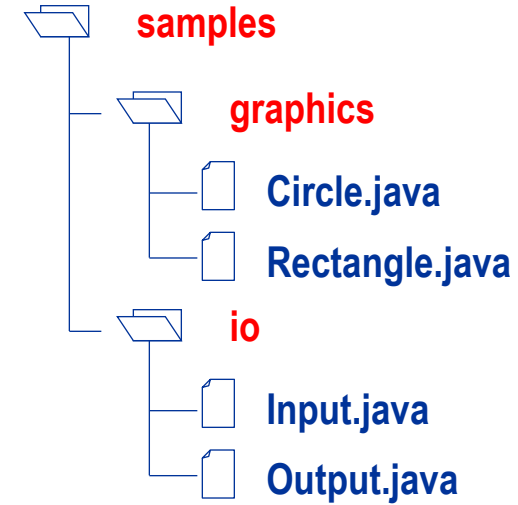
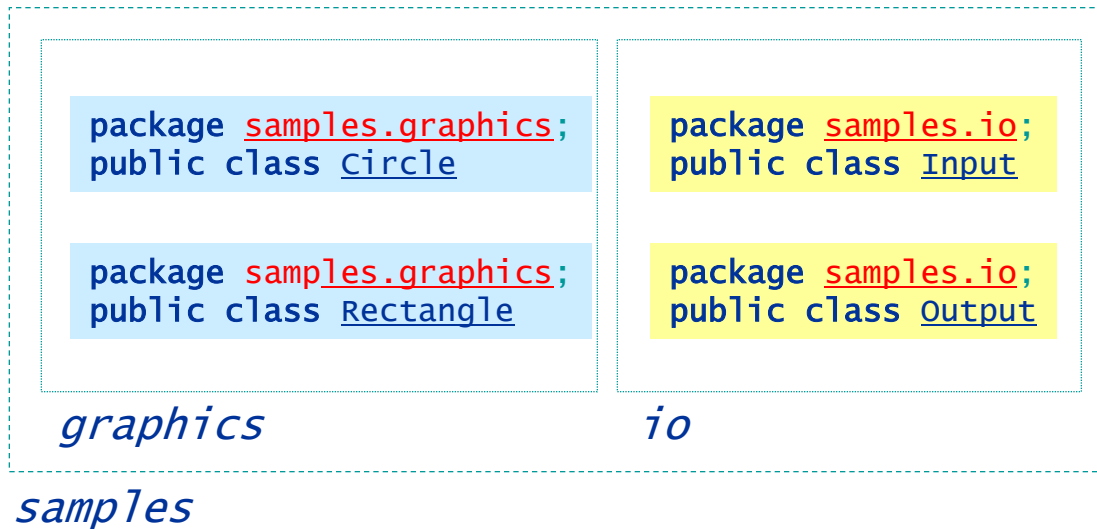
Übersetzung und Ausführung mit dem JDK

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

```
java P/A  
java P.A
```

} beides möglich

Pakete können zu größeren Paketen zusammengefaßt werden



Benutzung

```
import samples.graphics.Circle;
import samples.graphics.*;
import samples.*;

samples.io.Output out;
```

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



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

```
package one;  
class C {...}  
class D {...}
```

```
package two;  
class D {  
    C obj;  
}
```

← Compiler meldet einen Fehler!
C ist hier nicht sichtbar

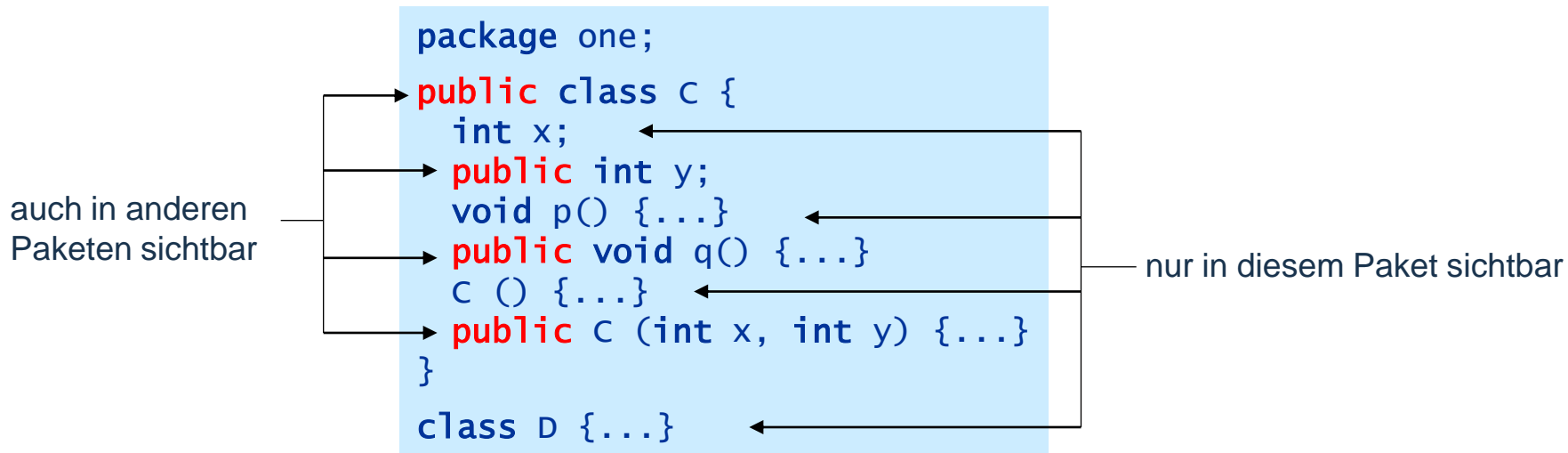
← gleicher Name stört nicht

Zweck

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

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.

Import von Klassennamen

Exportierte Klassennamen können in anderen Paketen importiert werden

Durch gezielten Import der Klasse

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

Durch Qualifikation mit dem Paketnamen

```
package myPack;
class MyClass {
    graphics.Circle c1;
    java.awt.Circle c2;
    one.C c;
    ...
}
```

Durch Import aller public-Klassen eines Pakets

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



- Zugriff auf Variablen und Methoden kann durch „Modifikatoren“ (visibility modifiers) geregelt
 - **public**
globaler Zugriff
 - **private**
Zugriff nur innerhalb der Klasse
 - **protected**
Zugriff innerhalb der Klasse, in allen Unterklassen und in allen Klassen des selben Pakets
 - **private protected**
Zugriff innerhalb der Klasse und in allen ihren Unterklassen
 - **Default (kein Modifier)**
Zugriff innerhalb der Klasse und in allen Klassen und Unterklassen des selben Pakets

Veranschaulichung Sichtbarkeitsattribute

für Felder und Methoden

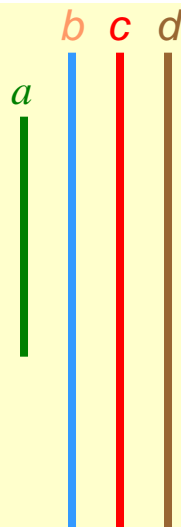
```
private int a;  
    int b;  
public int c;  
protected int d;
```

nur in der Klasse sichtbar, in der das Element deklariert wurde
nur im Paket sichtbar, in dem das Element deklariert wurde
auch in anderen Paketen sichtbar, wenn importiert
sichtbar:

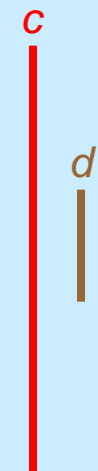
- in der deklarierenden Klasse
- in deren Unterklassen
- im deklarierenden Paket

Vererbung und
Unterklassen in SWE2

```
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 {  
    ...  
}
```



- Die Vererbung von Variablen und Klassen wird ebenfalls durch die Sichtbarkeitsregeln geregelt
 - **public**
Vererbung an alle Unterklassen (extends)
 - **private**
keine Vererbung
 - **protected**
Vererbung an alle Unterklassen
 - **private protected**
Vererbung an alle Unterklassen
 - **Default (kein Modifier)**
Vererbung an alle Unterklassen des selben Pakets

Zusammenfassung Ausblick



- **Prozedurales Programmieren in Java**
 - Daten, Algorithmen, Programme
 - Kontrollstrukturen
 - Elementare Datentypen
 - Arrays
- **Programmwurf**
 - Algorithmen
 - Assertionen und Kommentare
 - Schrittweise Verfeinerung
- **Elementare objektorientierte Programmierung**
 - Klassen und Objekte
 - Mengenbildung
 - Dynamische Datenstrukturen
- **Ausblick auf**
 - Vererbung und polymorphe Operationen
 - Ausnahmen
 - Pakete



Klassen

- innere Klassen
- anonyme Klassen
- abstrakte Klassen
- Interfaces

Threads

Bibliotheken

- GUI (Swing, AWT)
- Collections (Listen, Hashtabellen, ...)
- Streams (Ein-/Ausgabe)
- Net und Web
- Reflection

Anwendungen

- Applets
- Servlets und Java Server Pages
- Java Beans

Java™ 2 Platform, Standard Edition v 1.4

