Informatik I – Tutorium I

"Strickmuster":

- Grundlegender Algorithmenentwurf
- Aktive und Passive Objekte
- Datenstrukturen, Datenbehälter und ADTs
- Aktive Objekte
- GUIs und Ereignissteuerung
- Ströme

Typen und Vererbung:

- Vererbung als Wiederverwendung
- Typen und Typhierarchien
- Polymorphie

Grundlagen des Algorithmenentwurfs

- Programme arbeiten auf Daten
- Datentypen, strenge Typisierung:
  - Interpretation der gespeicherten Daten ist unveränderlich
  - Operationen typgebunden
- Grundtypen, "primitive Typen"
  - unmittelbare Maschinendarstellung
  - unmittelbare Darstellung im Programmtext (Literale)
  - intuitve Interpretation
  - v.a. Zahlentypen
  - zugehörige Operatoren

←→ Objekttypen, "benutzerdefinierte Typen"

Expression und Statement

- Expression: ***+ Funktion***
  - "Formel" – besitzt Auswertungsregel (Richtung, Prioritäten...)
  - hat Typ, liefert Wert
  - kann rechts von Zuweisung stehen

- Statement: ***+ void-Methode***
  - Anordnung, Ausführungsbefehl
  - Typ void, liefert keinen Wert
  - Verändert Zustände

Kontrollstrukturen

- Verzweigung:
  - if – else
  - if – else if – else — Kaskadiert, gut lesbar
  - switch – case (break!)
- Schleife
  - while — Bedingungsgebunden
  - do – while
  - for — Zählschleife
- Sprung
  - continue: — zum nächsten Schleifendurchgang
  - break: — hinter die Schleife / Kotrollstruktur
  - return: — aus der Methode heraus

Algorithmen-Gliederung

- Kontrollstrukturen
  - niemals tiefer als 3 schachteln,
  - 2 anstreben
- Schleifen
  - for für indizierte Struktruren
  - while für bedingte Abläufe oder Endlosschleifen

- Methoden
  - niemals länger als 15 Zeilen,
  - 5-8 anstreben
- Zerlegung
  - durch prozedurale Abstraktion (bottom-up)
  - oder schrittweise Verfeinerung (top-down)

Schleifen auf indizierten Strukturen

- for-Schleife für einfache indizierte Datenstruktur

```
String text = "....";
for (int i=0; i<text.length(); i++)
    writer.write(text.charAt(i));

char[] text1 = new char[20];
for (int i=0; i<text1.length; i++)
    text1[i] = i+0x41;

Vector text2 = new Vector();  // Elementtyp String
for (int i=0; i<text2.size(); i++)
     System.out.println(text2.elementAt(i));
```

Schleifen auf indizierten Strukturen

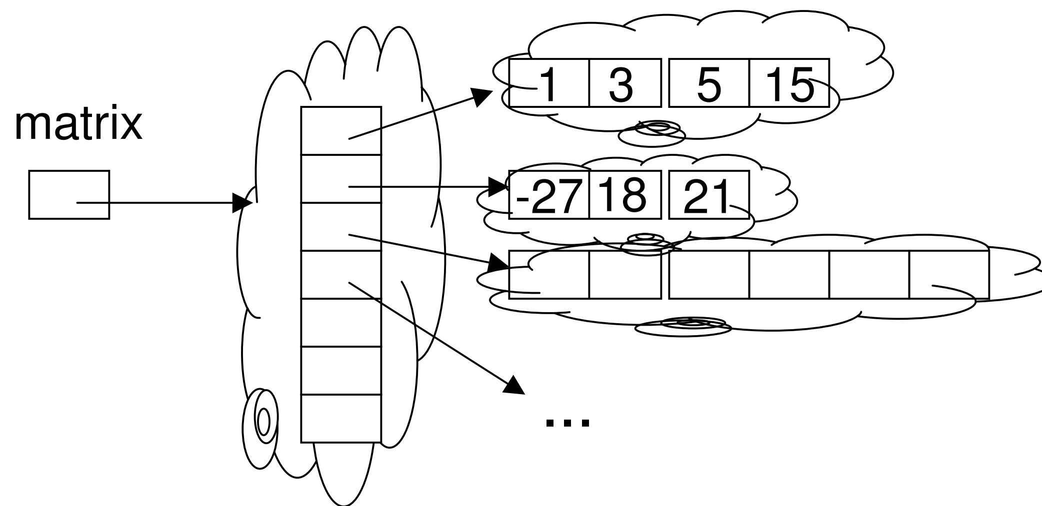
- for each-Schleife für einfache indizierte Datenstruktur

```
Vector<String> text2 = new Vector <String>();
  for (String s : text2)
     System.out.println(s);
                    // lesender Zugriff funktioniert

char[] text1 = new char[20];
  for (char c : text1)
    c = 'a';        // wirkungslos, c ist Kopie!!!
```

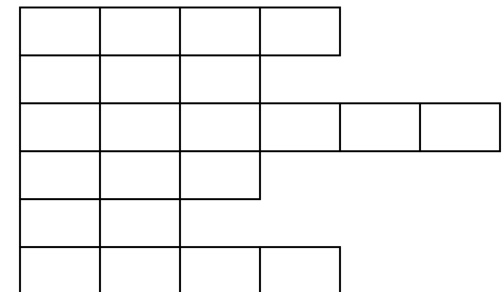
Geschachtelte Datenstrukturen

Datenstrukturen, deren Elemente wieder Datenstrukturen sind:

```
Vector[] listen;   // Array von Vectoren
int[][] matrix;    // 2-dim.Array: Array von int-Arrays
InputChannelVector in;  // vermutl. Vector von Channels
```



korrekte Datenstruktur



vereinfachtes Modell

Durchlaufen der Elemente

Zum Durchlaufen der Elemente braucht man geschachtelte Schleifen:

```
public static void initialize(Vector[] listen, Object val){
    for (int i=0; i<listen.length; i++)  // Array-Schleife
        for (int j=0; j<listen[i].size(); j++) // Vector-Schl.
            listen[i].elementAt(j)= val;
}
```

```
public static void initialize(int[][] matrix, int val){
    for (int i=0; i<matrix.length; i++)  // 1.Array-Schleife
        for (int j=0; j<matrix[i].length; j++) // 2.Array-Schl.
            matrix[i][j] = val;
}
```

```
public static void print(int[][] matrix){
    for (int[] zeile : matrix)  {// 1.Array-Schleife
        for (int element : zeile) // 2.Array-Schl.
            System.out.print(element + "\t");
        System.out.println();
}
```

Immer dasselbe "Strickmuster"

- Indextabelle

```
String[] aphorismen =...;
int[] indextable = new int[aphorismen.length];
for (int i=0; i<indextable.length; i++)
       indextable[i] = i;
```

Immer dasselbe "Strickmuster"

- Indextabelle

```
String[] aphorismen =...;
int[] indextable = new int[aphorismen.length];
for (int i=0; i<indextable.length; i++)
    indextable[i] = i;
```

- Matrix

```
int[][]zimmer = new int[etagen][raeume];
for (int i=0; i<zimmer.length; i++)
    for (int j=0; j<zimmer[i].length; i++)
        zimmer[i][j] = 10*i + j;
```

- Lesender Zugriff auch mit foreach

```
int[][]zimmer = new int[etagen][raeume];
for (int[] etage : zimmer)
    for (int raum : etage)
        System.out.println(raum);
```

3-dimensionales Array

? Funktioniert die Methode nur für feste Dimensionsgrößen (Quader) oder auch für freie Dimensionsgrößen (Stelenfeld)?

```
public static void initialize(int[][][] matrix, int val){
  for (int i=0; i<matrix.length; i++)  // 1.Array-Schleife
    for (int j=0; j<matrix[i].length; j++) // 2.Array-Schl.
       for (int k=0; k<matrix[i][j].length; k++) // 3.Array-Schl.
          matrix[i][j][k] = val;
}
```

Zugriffe auf geschachtelten Datenstrukturen

- ..nix anderes!

```
String[]texte1;
char[][]texte2;
Vector  texte3;   // Elementtyp String

// 2. Buchstabe des 12. Textes:
texte1[11].letterAt(1);
texte2[11][1];
((String)(texte3.elementAt(11))).letterAt(1);
```

Beispiel Encryption

```
String text;
char[]encryted;
int[] code1, code2;

encrypted[i] =
   text.letterAt(i) * code1[i+1+code2[i]];
```

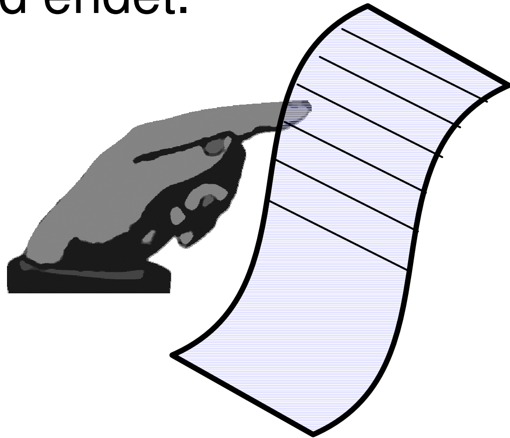
m	y	w	o	r	d	i	s	l	o	text
12	7	9	8	2	7	3	9	11	4	code1
2	3	7	2	1	3	11	8	6	7	code2
	242									encrypted

y * code1[1+1+3] == 121 * 2 == 242 small letter glottal stop

Zwei grundlegende Programmtypen

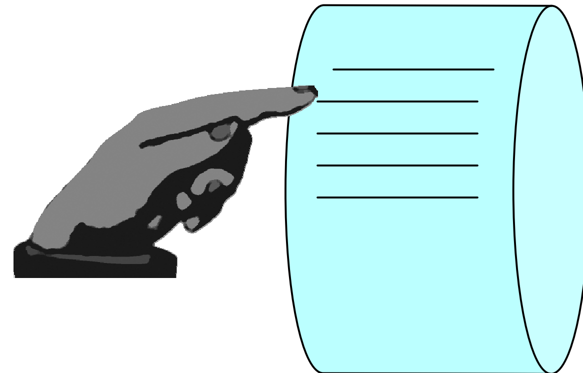
Sequentielles Programm
(Batch-Programm)

- wird (mit Eingabedaten) gestartet,
- läuft einmal durch,
- liefert ein Ergebnis
- und endet.



Interaktives Programm

- wird gestartet,
- läuft immer weiter,
- erwartet Anfragen,
- läuft für jede Anfrage einmal durch und wartet dann wieder.



Zwei grundlegende Kommunikationsformen

synchrone Kommunkation

asynchrone Kommunkation

Methodenaufruf:

- Programmausführung wird an der Aufrufstelle unterbrochen,
- Ausführung der Methode wird eingeschoben,
- Programmausführung wird fortgesetzt

Nachrichtenkanal:

- ein Objekt steckt etwas hinein
- ein anderes Objekt holt es bei Gelegenheit heraus

Zwei grundlegende Objekttypen

Passives Objekt

- hat keinen eigenen Thread
- bietet Methoden als Dienste an
- Methoden werden vom aufrufenden Thread ausgeführt
- kommuniziert mit anderen Objekten per Methodenaufruf

Aktives "animiertes"Objekt

- hat einen eigenen Thread
- oft als Endlosschleife
- Hauptmethode wird unanbhängig von anderen Objekten ausgeführt.
- kommuniziert mit anderen Objekten per Methodenaufruf (synchron)
- kommuniziert mit anderen aktiven Objekten typischerweise über einen Kanal (asynchron)

Die Startklasse eines Programms ist aktiv:
Der Thread der virtuellen Maschine ruft die main-Methode

Grundlegende Programmstruktur

Main-Methode erzeugt ein oder mehr Problemlösungs-Objekte

passive Objekte:
main ruft Methoden der Objekte (teilt seinen Thread mit ihnen), um die Leistung abzurufen und zu verwenden.

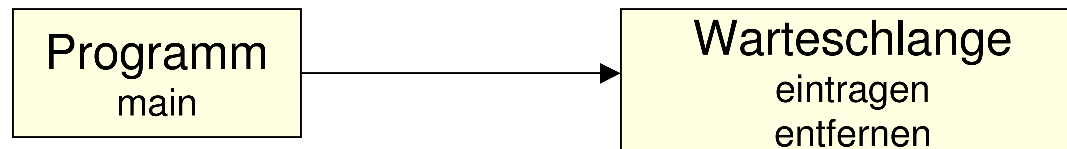
Main hat seine Leute, alles läuft von selbst.

Main ist Chef, hat alles unter Kontrolle...

aktive Objekte:
mit der Objekterzeugung (+ ggf. Thread-Zuordnung) beginnen die Objekte selbständig mit der Arbeit, main ist fertig.

Main für passives Objekt

```
public class Programm {
  public static void main(String[] args) {
     Warteschlange schlange = new Warteschlange();
     schlange.eintragen(new Patient());
     schlange.eintragen(new Patient());
     try {
       Console.println(schlange.entnehmen().toString());
       Console.println(schlange.entnehmen().toString());
       Console.println(schlange.entnehmen().toString());
     } catch (LeerException ex){Console.println("leer");}
   }
}
```



Exceptions:
Nicht jeder Auftrag ist ausführbar

Manchmal stimmen die Voraussetzungen für den Methodenaufruf nicht:

```
miraculix.zaubertrankBrauen();
```
?

Bekanntlich weist Miraculix diesen Auftrag oft zurück, z.B. weil kein Neumond ist...

```
try {miraculix.zaubertrankBrauen(); }
 catch (MondException m)
           { frustriertGehen(); }
```

In der Klasse Druide gäbe es dann folgende Definition:

```
class Druide {
      public void zaubertrankBrauen() throws MondException
      {  if (!neumond()) throw new MondException();
         /* der Rest bleibt geheim ... */
      }
}
```

Exceptions

- definieren

```
class MyException extends Exception();
```

- auswerfen und deklarieren

```
public void method1() throws MyException {
    if (x>y) throw new MyException();
}
```

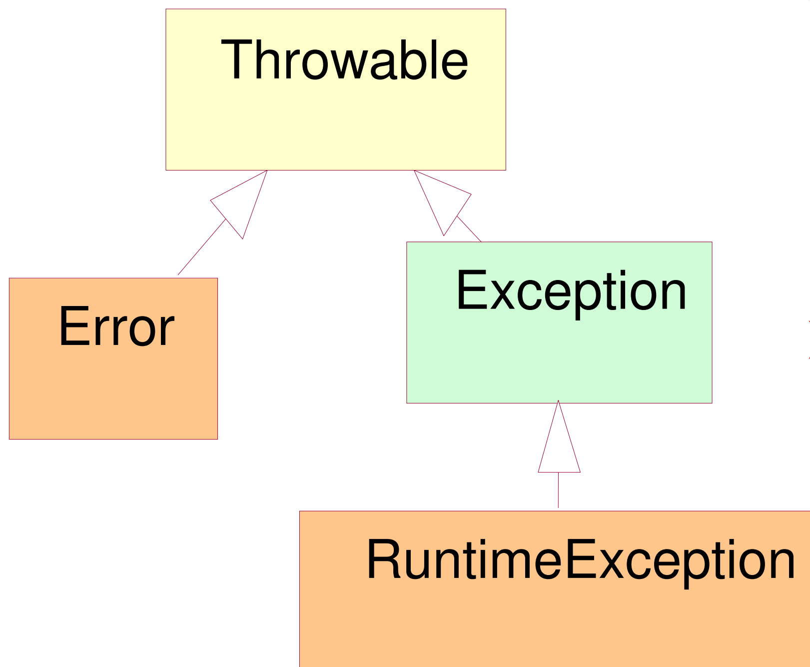
- an der Aufrufstelle fangen

```
public void method2() {
  try { method1(); }
  catch (MyException ex){ es.printStackTrace(); }
}
```

- oder weitergeben

```
public void method3() throws MyException
{    method1();    }
```

Exception-Typhierarchie



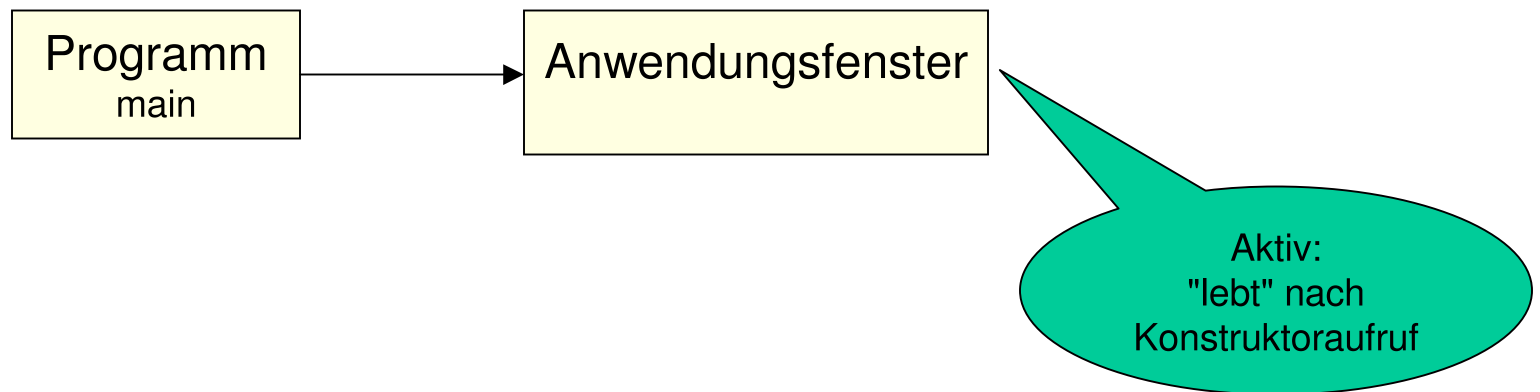
- Exceptions- are to be used for repairable Errors.
  They must be either caught or passed on through specification.
  → checked exceptions

- Errors –are meant for system errors; typically they cannot be handled.
  Your program may catch them, but does not need to.

- RuntimeExceptions- are meant for program errors that cannot be repaired.
  Your program may catch them, but does not need to.

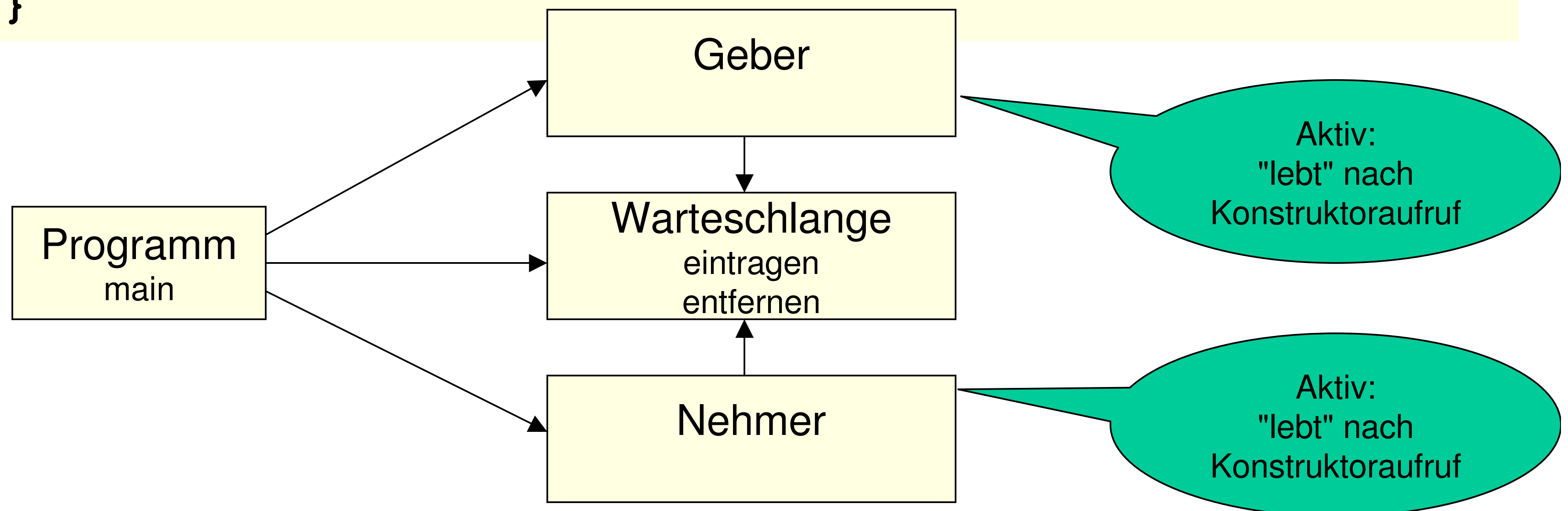
Main für aktives Objekt (GUI):

```
public class Programm {
  public static void main(String[] args) {
     JFrame fenster = new Anwendungsfenster();
  }
}
```



Main für aktive Objekte (Nachrichtenkanal)

```
public class Programm {
  public static void main(String[] args) {
     Warteschlange schlange = new Warteschlange();
     Geber geber = new Geber(schlange);
     Nehmer nehmer1 = new Nehmer(schlange);
     Nehmer nehmer2 = new Nehmer(schlange);
  }
}
```



Aufgabe:
Aktives Objekt implementieren

- "Lebendgeburt":
  Thread im Konstruktor erzeugen und zuordnen.

- Möglichkeit schaffen festzulegen, auf welchen anderen (passiven) Objekten das Objekt operiert:
  – Typischerweise durch Übergabe im Konstruktor (Konstruktorparameter)
  – Alternativ durch eine "set"-Methode

Beispiel: Füllstandsbeobachtung

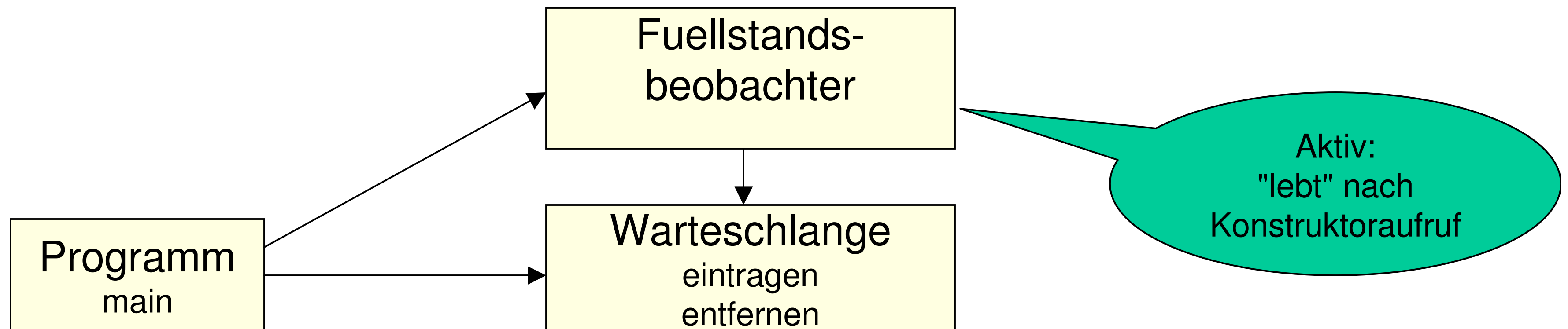
- Schreiben Sie ein aktives Objekt, das eine Warteschlange "beobachtet", indem es regelmäßig ihren Füllstand liest und ausdruckt.
- Die zu beobachtende Warteschlange wird im Konstruktor übergeben.

Beispiel: Füllstandsbeobachtung

```
public class Fuellstandsbeobachter implements Animate {
   private Warteschlange schlange;

  public Fuellstandsbeobachter(Warteschlange schlange) {
     this.schlange = schlange;
     AnimatorThread lebensfaden = new AnimatorTread(this);
     lebensfaden.startExecution();
  }
  public void act() {
     Console.println("Fuellstand: "+schlange.anzahl());
  }
}
```

Aktives Objekt starten (testen)

```
public class Program {
  Warteschlange schlange = new Warteschlange();
  Fuellstandsbeobachter checkIt =
     new Fuellstandsbeobachter(schlange);
```



Wann braucht man aktive Objekte?

- Vereinfachter Kontrollfluss:
  - keine künstliche Sequentialisierung unabhängiger Vorgänge
- Blockade-Vermeidung:
  - Wenn Blockade oder langes Warten droht
  - Synchrone Kommunikation durch aktives Objekt und asynchrone Kommunikation ersetzen
- Zwischenzustände kapseln:
  - Wenn Aufbau eines stabilen neuen Zustands Zeit benötigt
  - Aufbau in aktives Objekt verlagern, Übernahme erst nach Fertigstellung
  - Beispiel: ImageObserver
- Beobachten ohne zu "stören"
  - Steuernder Prozess läuft, ggf. zeitkritisch
  - Visualisierung durch eigenständigen Beobachter-Thread
  - Kopplung asynchron (über vorhandene Daten)

Beispiel Guitar Tuner (simplified)

```
public class GuitarTuner implements Animate {

   private Guitar guitar;
   private Display display;
   int targetPitch = 440;  // default is "a"

   public GuitarTuner (Guitar guitar){
        display = new Display();
        this.guitar = guitar;
        AnimatorThread thread = new AnimatorThread(this);
        thread.startExecution();
  }

  public void act() {
        int diff = guitar.getPitch()- targetPitch;
        if (diff < 0) display.show("up");
        else if (diff > 0) display.show("down");
        else display.show("fine");
    }

  public void setPitch(int pitch)
  { this.targetPitch = pitch; }
}
```

Main für Gitarrenbeispiel

```
public static void main(String[]args) {
  Guitar myguitar = new Guitar();
  guitar.startPlaying();
  GuitarTuner tuner = new GuitarTuner(myguitar);
  tuner.setPitch(440);
}
```

Datenstruktur, Datentyp, ADT

- Datenstruktur:
  - Objekt zum Speichern von Daten
  - Beispiel `String[]passphrases`
- Datentyp:
  - Klasse oder Typ zur Definition von Datenstrukturen
  - Beispiele:

```
int[]
public class Address {
  String name;
  String firstName;
  int age;
  // …
}
```

- ADT (Abstrakter Datentyp)
  - Klasse mit einer privaten Datenstruktur
  - öffentlichen Zugriffs- und Manipulations-Methoden
  - Zugriffs-Strategie

Aufgabe: Abstrakten Datentyp implementieren

- ADT typischerweise durch ein Interface definiert.
- Implementierung:
  - Klasse mit einem privaten Array oder einer anderen privaten Datenstruktur
  - Vom Interface geforderte Methoden mithilfe der privaten Datenstruktur implementieren

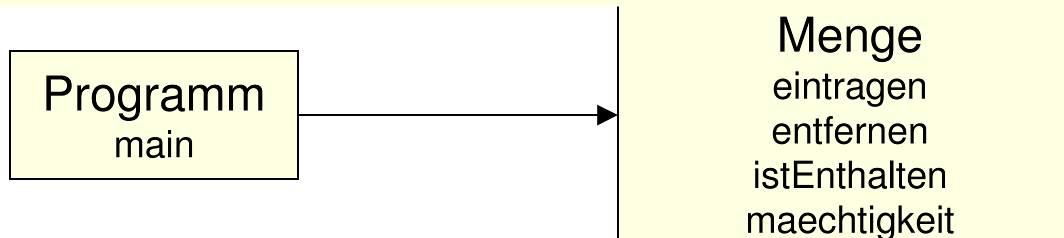
Beispiel Menge

```
public interface MengenInterface{
        public void eintragen(Object obj);
                // eintragen, falls noch nicht vorhanden
        public void entfernen(Object o)
           throws NichtEnthaltenException;  // entnehmen
        public boolean istElement(Object o);
        public int maechtigkeit();
}
```

```
public class Menge implements MengenInterface {
        private int groesse = 0;
        private ArrayList liste = new ArrayList();

        public void eintragen(Object obj) {
                if (!liste.contains(obj)){
                    liste.add(obj);
                    groesse++;
                }
        }
        // etc
}
```

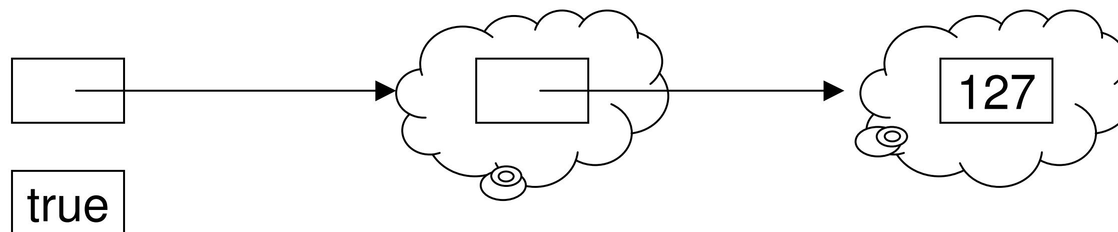
Datenbehälter benutzen (testen)

```
public class Programm {
  public static void main(String[] args) {
     Menge menge = new Menge();
     menge.eintragen("Cola");
     menge.eintragen("Cola");
     try {
       menge.entfernen("Cola");
       Console.println("Cola entfernt");
       menge.entfernen("Cola");
       Console.println("Cola entfernt");
     } catch (NichtenthaltenException ex)
       {Console.println("nicht enthalten");}
  }
}
```

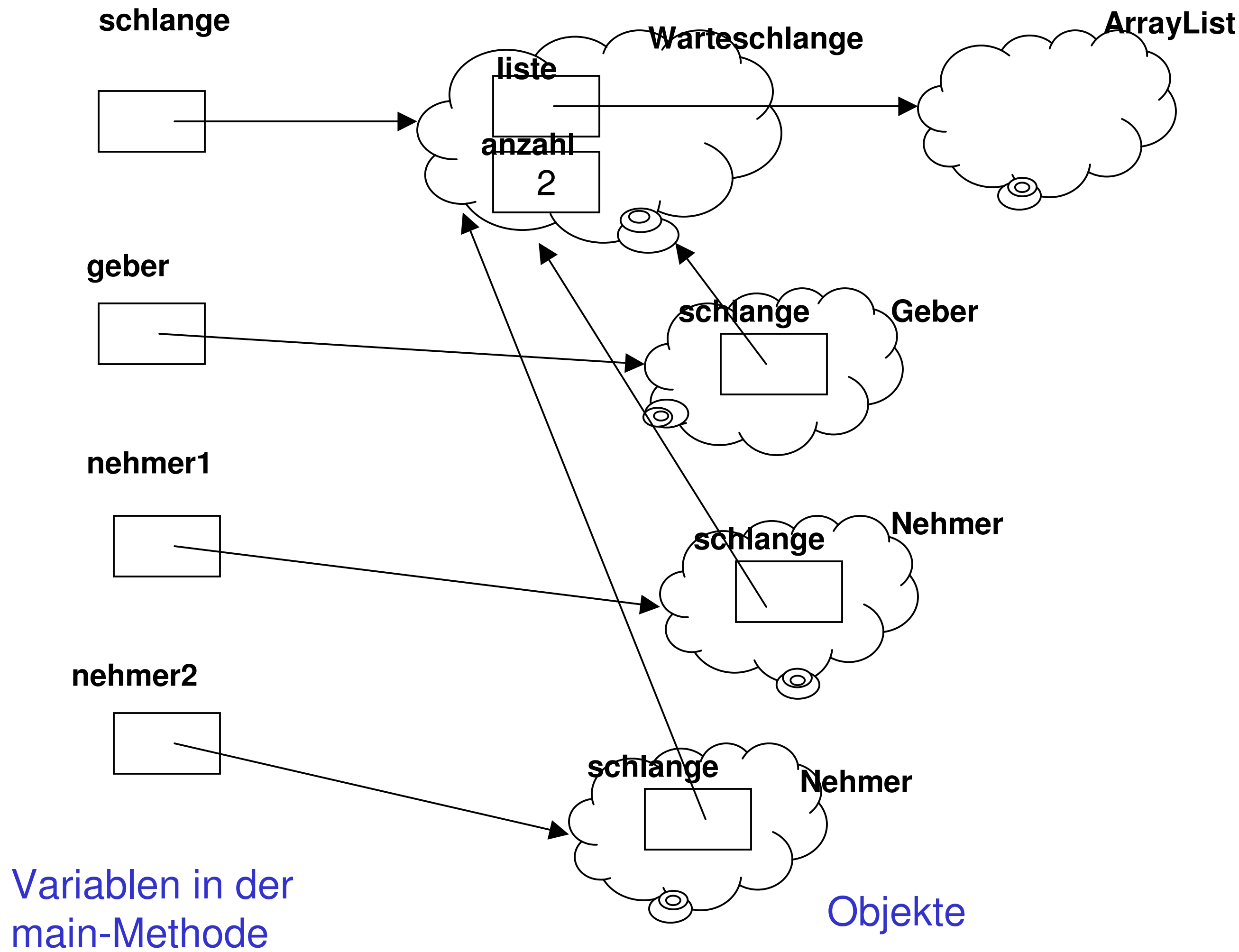


Datenstrukturen verstehen und zeichnen

- "Schnappschuss" zur Laufzeit
- Welche Variable / welches Attribut enthält welchen Wert?
- Variable/Attribut: Kästchen
- Primitiver Wert: Eintrag im Kästchen
- Objekt-Wert: Pfeil (Referenz) auf eine Wolke (Objekt)
- Wolke enthält wieder Kästchen (Attribute)



Datenstruktur zum Nachrichtenkanal (Testmethode)



Variablen in der main-Methode

Objekte

Aufgabe: GUI implementieren

- Klasse definieren, die (z.B.) JFrame erweitert
- Feste (Hintergrund-)Grafik durch Überschreiben von paint() erstellen.

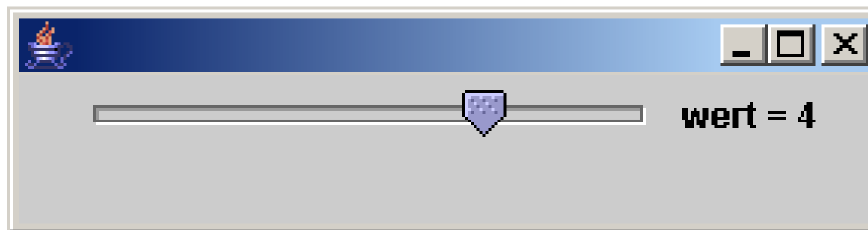
- init-Methode schreiben, die alle Widgets erzeugt und dem Fenster hinzufügt
- Im Konstruktor init(), setDefaultClosingOperation(), setSize() rufen. Zum Schluss setVisible(true) rufen.

- In der init-Methode den Komponenten (Widgets, Container) Ereignisbehandler hinzufügen
  - entweder als anonyme Ereignisbehandler:
    Definition und Instanziierung im Parameter von addXXListener
  - oder als "this" (Universal-Behandler):
    Klasse implementiert Listener-Interface
    Behandlermethoden sind in der Klasse definiert
    aktivieren durch *this.addXXListener(this)* – nicht vergessen!!

Beispiel: Slideranzeige

```
import java.awt.*;import javax.swing.*;import javax.swing.event.*

public class SliderWindow extends JFrame
                          implements ChangeListener {
      private JLabel wert;
      private JSlider slider;

      private void init() {
            slider = new JSlider(-10,10,0);
            wert = new JLabel("wert = 0");
            this.setLayout(new FlowLayout());
            this.add(slider);
            this.add(wert);
            slider.addChangeListener(this);
      }

      public SliderWindow() {
            init();
            setVisible(true);
      }
```

```java
        /* Listener-Methode */
        public void stateChanged(ChangeEvent arg0) {
                wert.setText("wert = "+slider.getValue());
        }

        public static void main(String[] args) {
                new SliderWindow();
        }
}
```




```java
/* Alternative: anonymer Ereignisbehandler */
slider.addChangeListener(new ChangeListener(){
        public void stateChanged(ChangeEvent arg0) {
                wert.setText("wert = "+slider.getValue());
}});
```

Zweistromland java.io

- Stream-Typen
  lesen/schreiben entweder byte oder char:
- 4 abstrakte Klassen

InputStream	**Reader**	read() close()
OutputStream	**Writer**	write() close()
MsgType = byte	MsgType = char	

- ALLE Methoden werfen IOException
- Unterklassen für verschiedene Aufgaben

Arbeiten mit Strömen

1. Stromquelle/-senke erschließen

```
InputStream input = System.in;
InputStream fileinput =
             new FileInputStream("myfile.xx");
OutputStream output = socket.getOutputStream();
```

2. Stromform anpassen

```
Reader inreader = new InputStreamReader(input);
BufferedInputStream lineinput =
            new BufferedInputStream(fileinput);
BufferedWriter linewriter =
      new BufferedWriter
         (new OutputStreamWriter(output));
```

Stromschnellen

- read() blockiert
  - möglichst nicht synchron benutzen
  - im eigenen Thread rufen
  - Ergebnis asynchron hinterlegen
- write() versendet blockweise
  - abschließend flush() rufen, um den Rest zu versenden
- Ströme besetzen OS-Ressourcen
  - mit close() schließen

Kommunikatoren

- Klassen, die die Strom-Ein-Ausgabe kapseln
- Exportieren Lese- und Schreibmethoden
- implementieren "Stromschnellen-Umschiffungen"
- können Protokolle umsetzen

II. Typen und Vererbung

Vererbung als Wiederverwendung

- Sie wollen Teile Ihrer Klasse einer fertigen Klassen "nachempfinden"?
- Copy&Paste hat Nachteile:
  - Kopierfehler
  - Redundanter Code etc.
- Erweitern heißt die "High Tech"-Lösung:
  - Kein redundanter Code
  - Kein Quellcode erforderlich

Beispiel Stack mit "Einsicht"

```
// Quellcode nicht bekannt, wohl aber Dokumentation

public class InspectableStack extends Stack {

  public Object elementAt(int i)
                  throws IndexOutOfBoundsException {
      return list.elementAt(i);
                // list muss protected sein!
                // Ohne Dokumentation geht es nicht...
  }
}
```

Vererbung zur Typdefinition

- Sie wollen eine verbesserte Klasse schreiben, die anstelle der alten Klasse verwendet werden kann?
- Die Unterklasse hat (auch) den Typ der Oberklasse
- D.h. Unterklassenobjekte können jederzeit wie Oberklassenobjekte benutzt werden.
- Umgekehrt geht es nicht – der Unterklassentyp kann Methoden enthalten, die die Oberklasse nicht hat, also für Oberklassenobjekte nicht aufrufbar sind.

```
public class Rechenwerk {
   private Stack arbeitsspeicher;

   public Rechenwerk(Stack stack) {
      this.arbeitsspeicher = stack;
   }
   public void enterNumber(Number num) {
       arbeitsspeicher.push(num);
   }
   public void enterOperator(Operator op) {
       if (arbeitsspeicher.anzahl == 1) push(op);
       else {
          push(doOperation(pop(),pop(),pop());
          push(op);
       }
   }
}
```

```
public static void main (String[] args)
{  Rechenwerk calc = new Rechenwerk (new InspectableStack()); }
```

Typanpassung

- Zuweisung an eine Variable als Denkmodell
  - `ATyp a; BTyp b; a=b; //erlaubt??`
- primitive Typen:
  - Alle Zahlentypen sind aneinander anpassbar
  - geschieht durch Umspeichern des Wertes
  - Automatische Anpassung, wenn kein Info-Verlust droht ("widening")
  - Sonst explizite Anpassung ("Casting"), ggf. Info-Verlust
- Objekttypen:
  - Innerhalb einer Typhierarchie aneinander "anpassbar"
  - Implizite Anpassung an eine Oberklasse
  - Explizite Anpassung an ein Unterklasse
  - nur die Referenz wird angepasst
  - kein Umspeichern, kein Info-Verlust beim Objekt

Primitive Typen: Typ als Darstellungsform des Werts

- Typanpassung nur zwischen Zahlenwerten
  (nicht Zahlen, Zeichen, boolesche Werte).

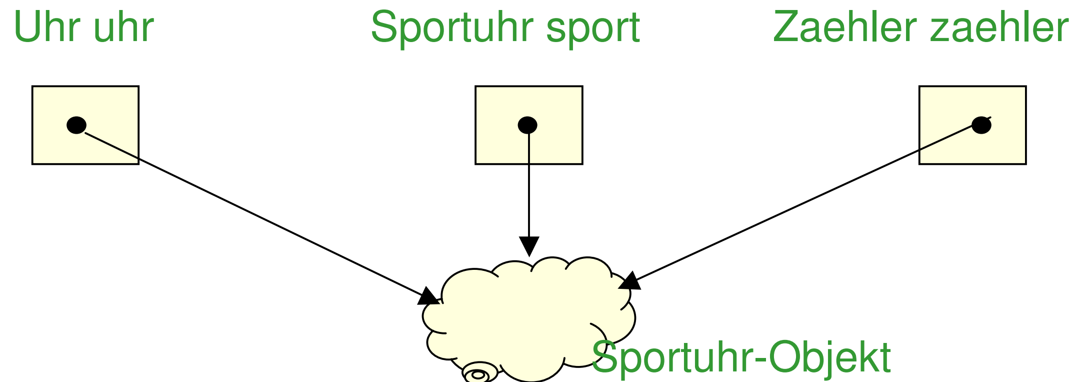
- Implizite Typhierarchie nach Genauigkeit:
  double → float → long → int → short → byte

- Implizite Typanpassung dort, wo Genauigkeit vergrößert wird:
  "widening" **`(double) 3 → 3.0`**

- Explizite Typanpassung dort, wo die Genauigkeit eingeschränkt wird:
  "narrowing" **`(int) 3.1415926 → 3`**

- Die Typanpassung ändert den Wert.
  → explizite Typanpassung:
  Information (hier die Nachkommastellen) wird nicht "verborgen", sondern geht verloren:
  `(double)((int) 3.1415926) → 3.0`

Objekttypen: Typ als Aspekt



Drei verschiedene Sichten auf dasselbe Objekt

```
Sportuhr sport = new Sportuhr();
Zeit zeit = sport.zeitLesen();    // ok
sport.zaehlen();                  // ok
sport.stoppStart();               // ok

Uhr uhr = sport;                  // ok
zeit = uhr.zeitLesen();           // ok
uhr.zaehlen();                    // Typfehler
uhr.stoppStart();                 // Typfehler

Zaehler zaehler = sport;          // ok
...
```

Typanpassung bei Objekttypen

- Typanpassung, englisch:
  - "Coercion" (Vergewaltigung),
  - "Casting" (eine Rolle zuordnen)

- Implizite Typanpassungen sind immer möglich,
  wenn nur ein Aspekt benötigt wird:

```
Uhr uhr = new Sportuhr();
```

- Explizite Typanpassungen sind möglich, wenn das Objekt mehr bietet als der Typ des Namens verrät:

```
((Sportuhr)uhr).zaehlen();// Aspekterweiterung
```

- Sie geschehen "auf eigene Gefahr" und können scheitern:

```
uhr = new Uhr();
((Sportuhr)uhr).zaehlen();// ClassCastException!!
```

- Typanpassungen können beliebig hin und her vorgenommen werden.
  Das Objekt bleibt unverändert.

```
((Sportuhr)(Uhr)(Sportuhr)uhr).zaehlen();
```

Vererbung und Polymorphie

- Sie wollen eine verbesserte Klasse schreiben, die anstelle der alten Klasse verwendet werden kann?
- Die verbesserte Klasse soll sich in manchen Fällen (d.h. für manche Methoden) anders verhalten als die Oberklasse:
- Überschreiben → Polymorphie

Beispiel Stack mit "Protokoll"

```
// Quellcode nicht bekannt

public class VerboseStack extends Stack {

  public Object pop() throws LeerException {
     Object obj = super.pop();
     Console.println(obj.tostring());
     return obj;
}
```

Objekte des Typs VerboseStack verhalten sich genauso wie Objekte vom Typ Stack – außer bei pop()

```
public static void main (String[] args)
{  Rechenwerk calc =
             new Rechenwerk (new VerboseStack()); }
```

Das waren sie: die großen Themen des Semesters

"Strickmuster":

- Aktive und Passive Objekte
- Datenbehälter implementieren und benutzen
- Aktive Objekte implementieren
- GUIs implementieren
- Datenstrukturen verstehen und zeichnen

Typen und Vererbung:

- Vererbung als Wiederverwendung
- Vererbung zur Typdefinition
- Vererbung und Polymorphie

Nächstes Mal üben wir konkret für die Klausur.