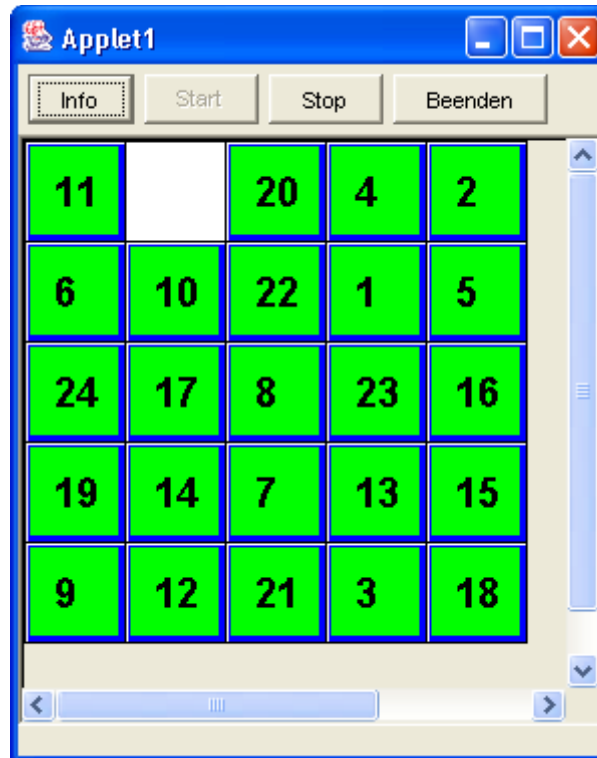


Einführung in die Informatik - Teil XII - Klassen und Objekte



Inhalt:

1. Klassen als Container

1.1 Verbunde

1.2 Beispiel: Schiebepuzzle

1.2.1 Die Klasse der Spielsteine

1.2.2 Das Fenster nutzt die Klasse der Spielsteine

1.3 Beispiel: Physikobjekte

1.3.1 Klassenvariable

1.3.2 Verschiebbare Geräte

1.3.3 Virtuelle Methoden und Polymorphismus

1.3.4 Kommunikation zwischen Objekten

1.3.5 Überladene Konstruktoren

1.3.6 Die Klasse der Gerate

1.3.7 Die Klasse der GerateMitThread

1.3.8 Die Klasse der Digitalanzeigen

1.3.9 Die Klasse der Digitalvoltmeter

1.3.10 Die Klasse der Buchsen

1.3.11 Aufgaben

Literaturhinweise:

- Küchlin/Weber: Einführung in die Informatik, Objektorientiert mit Java, Springer 1998
- Krüger, Guido: Handbuch der Java-Programmierung, <http://www.javabuch.de> oder Addison Wesley 2002

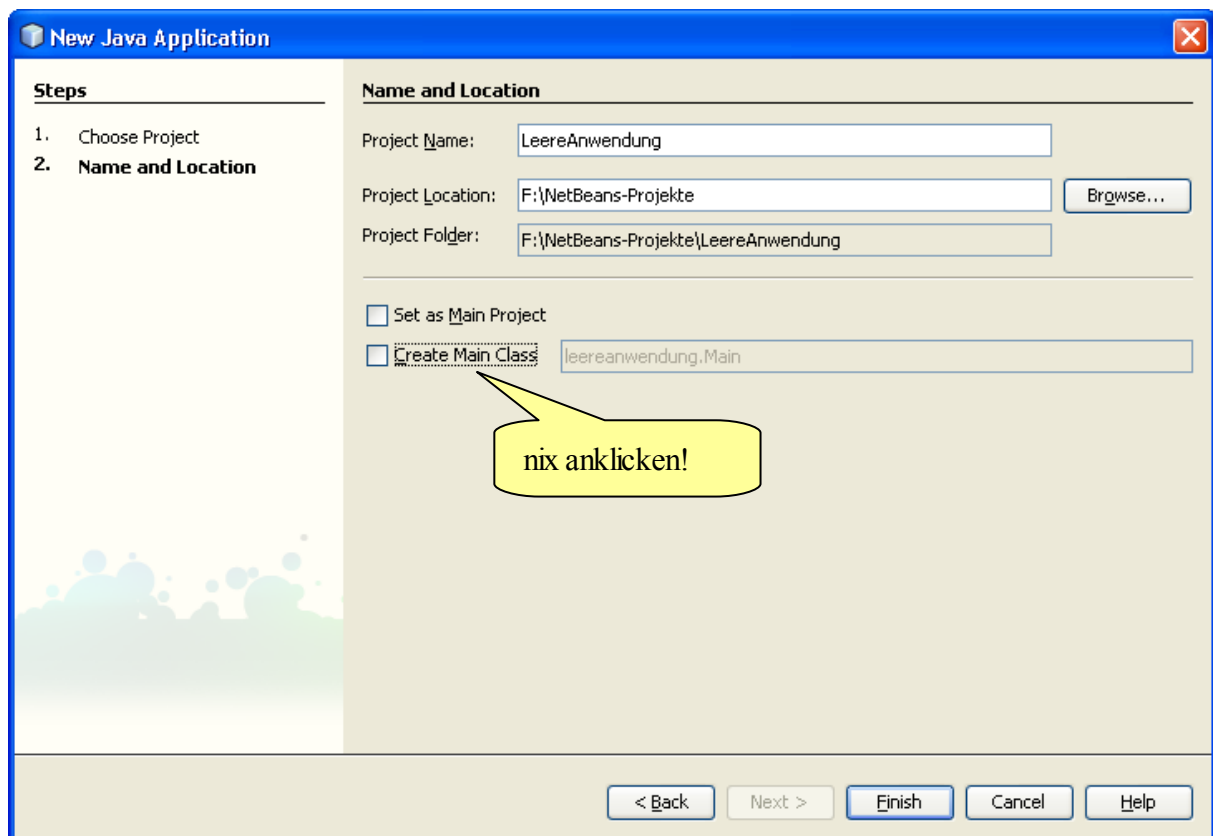
1. Klassen als Container

Java-Programme bestehen immer aus mindestens einer Klasse. Diese muss als *public* deklariert sein, um „von außen“ gestartet werden zu können. (Bei Applets geschieht das in der bekannten Aufrufreihenfolge durch einen Browser.) Fast immer enthalten solche „Programme“ eine Methode *main()*, die als *static* deklariert ist. Warum?

Statische Methoden sind *Klassenmethoden*, die unabhängig von der Existenz einer Instanz dieser Klasse existieren. Sie können also gestartet werden, bevor überhaupt ein Objekt dieser Klasse erzeugt worden ist – und genau dafür werden sie benutzt: in der *main()*-Methode werden fast immer Objekte neu erzeugt, insbesondere Instanzen der Klasse, die gerade „gestartet“ wurde. Diese Objekte können dann im Rechner agieren.

Meist enthalten Java-Klassen auch (mindestens einen) *Konstruktor*, der aufgerufen wird, wenn ein Objekt dieser Klasse durch den *new*-Operator erzeugt wird. Der Konstruktor muss den gleichen Namen wie die Klasse haben, als *public* deklariert sein und sonst keine weiteren Modifizierer enthalten.

Betrachten wir die rudimentäre Form einer Java-Anwendung, wie sie von NetBeans eingerichtet wird. Zuerst richten wir ein Projekt ein:



Dann sehen wir uns den leicht veränderten Code an. (Die Paketnamen wurden als Importe vorangestellt, weil sonst die Namen zu lang werden. Außerdem wurde die Fenstergröße eingestellt.)

Wir benötigen eine Klasse, die ein Fenster verwalten kann:

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

Importierte Pakete (s. Bemerkung oben)

```
public class LeererFrame extends Frame
```

eine Anwendung in einem eigenen Fenster wird geschrieben

```
{
    public LeererFrame()
    {
        initComponents();
        setSize(600,400);
    }
```

im **Konstruktor** wird eine Methode aufgerufen, die das Fenster „einrichtet“, dann wird dessen Größe eingestellt

```
private void initComponents()
```

```
{
    addWindowListener
    (
        new WindowAdapter()
        {
            public void windowClosing(WindowEvent evt)
            {
                exitForm(evt);
            }
        }
    );
    pack();
}
```

Ein anonymes Objekt ohne Namen wird als Parameter erzeugt.

Komponenten initialisieren (wird im Code „versteckt“) Der Code wurde etwas leserlicher umformatiert.

```
private void exitForm(WindowEvent evt)
```

```
{
    System.exit(0);
}
```

ggf. Fenster schließen

Das ist die reine „Bürokratie“ und könnte sehr viel einfacher geschrieben werden! Da der gesamte Code aber automatisch erzeugt und im Editor „versteckt“ wird, kann man damit leben.

Im gesamten Quelltext bis hierhin wurde noch gar nichts Sichtbares geschaffen, sondern nur beschrieben, **WIE** das ggf. zu geschehen hätte. **WO** also wird das Fenster erzeugt und die Anwendung gestartet?

Wir benötigen einen „Einspringpunkt“ ins Programm, eine Methode, die von der virtuellen Java-Maschine gestartet wird. Genau dafür ist die Methode *main(...)* vorgesehen. In dieser wird das Fenster mit dem aufgerufenen Konstruktor erzeugt – mit *new LeererFrame....* – Dann wird die Kontrolle an dieses abgegeben.

```
public static void main(String args[]) {
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            new LeererFrame().setVisible(true);
        }
    });
}
```

Das Fenster wird in der *main(...)*-Methode erzeugt

Das Fenster wird nun Objekte (Buttons, Textfelder, ...) enthalten, die auf Ereignisse reagieren. In den entsprechenden *Event-Handlern* wird Code stehen, der in seiner Gesamtheit die Funktionalität des Programms bildet.

1.1 Verbunde

Klassen können Felder und Methoden enthalten. Bei Feldern handelt es sich um Variable (und ggf. Konstante), die von einem beliebigen Typ sein können. Methoden entsprechen Unterprogrammen konventioneller Programmiersprachen. Klassen brauchen aber durchaus nicht beide Komponentenarten zu enthalten. Sowohl die Felder wie die Methoden können wegfallen.

Eine Klasse, die nur Felder enthält, kann also nur Daten aufnehmen. Sie entspricht damit dem Datentyp *Verbund* (Record, Struct, ..) anderer Sprachen. Normalerweise fasst man damit Daten unterschiedlichen Typs zusammen, die logisch zusammen gehören. (Gleichartige Daten kann man besser in Reihungen zusammenfassen.) Standardbeispiel für Verbunde sind Personaldaten:

```
class Personaldaten
{
    int    persNr;
    String name, vorname;
    String gebDatum;
}
```

Will man eine Firma modellieren, dann legt man z. B. eine Reihung von Mitarbeitern an:

```
Personaldaten[] AlleMeineMitarbeiter = new Personaldaten[20];
```

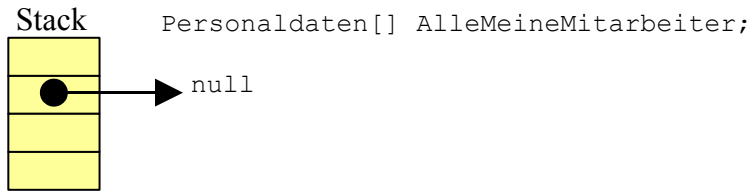
Steht der Mitarbeiter *Paul Meyer* an dritter Stelle in der Reihung, dann können seine Daten direkt eingegeben werden (natürlich erst, nachdem entsprechende Instanzen erzeugt wurden):

```
class test
{
    Personaldaten[] AlleMeineMitarbeiter = new Personaldaten[20];

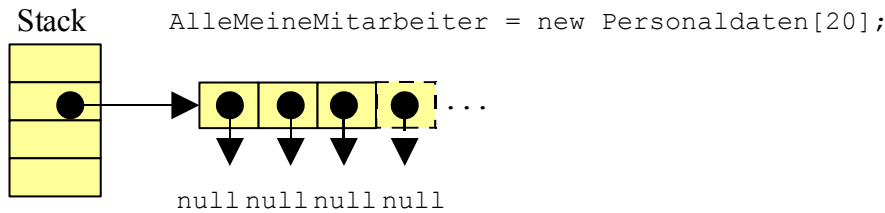
    public void arbeite()
    {
        for(int i=0;i<20;i++)
            AlleMeineMitarbeiter[i] = new Personaldaten();
        ...
        AlleMeineMitarbeiter[2].persNr    = 1234;
        AlleMeineMitarbeiter[2].name      = "Meyer";
        AlleMeineMitarbeiter[2].vorname   = "Paul";
        AlleMeineMitarbeiter[2].gebDatum  = "18.2.48";
        ...
    }
}
```

Betrachten wir die Sache mal aus der Sicht der Referenzen:

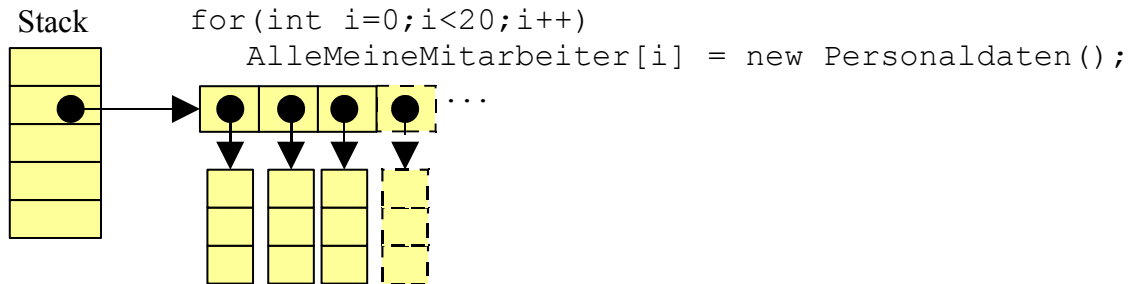
Zuerst wird eine Reihung von Personaldaten vereinbart ...



... und danach sofort instantiiert durch den Aufruf des Konstruktors:



Zuletzt werden die Personaldaten-Objekte der einzelnen Mitarbeiter instantiiert:



Erst danach ist wirklich Platz für Personaldaten vorhanden.

```
AlleMeineMitarbeiter[2].persNr = 1234;
...
```

Aufgaben:

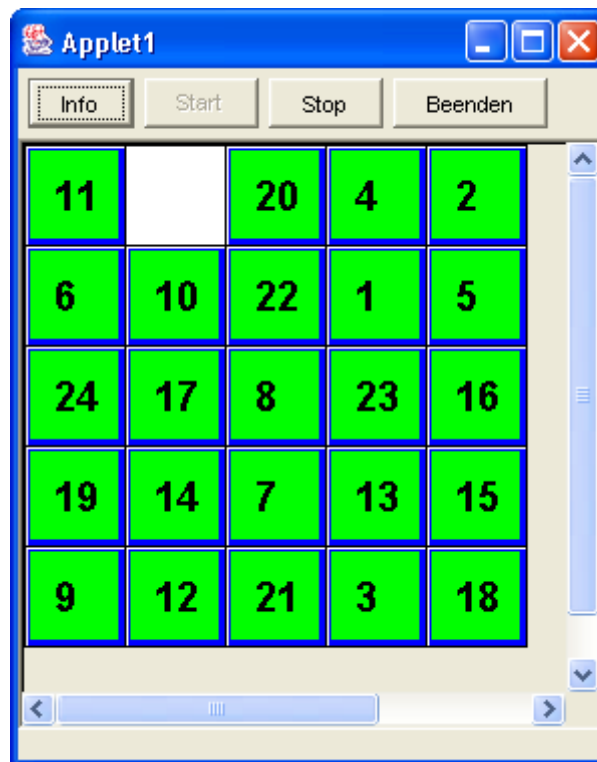
1. Verwalten Sie die Daten von mehreren Grafikobjekten:
 - a: Führen Sie eine Klasse Grafikdaten ein, die nur Felder enthält, die Grafikobjekte beschreiben (Position, Farbe, Art des Objekts, ...)
 - b: Führen Sie ein Feld, das mehrere dieser Objekte aufnimmt. Füllen Sie das Feld „per Hand“ (als im Programmtext) mit Werten.
 - c: Veranschaulichen Sie sich die Aktionen auf Stack und Heap anhand von Skizzen (s.o.).
 - d: Stellen Sie die Objekte am Bildschirm dar, sowohl innerhalb eines Applets wie auch in einer Anwendung. Beachten Sie die unterschiedlichen zu Grunde liegenden Grafikmodelle.
 - e: Lassen Sie die Grafikobjekte am Bildschirm interaktiv erzeugen. Speichern Sie die beschreibenden Größe in Ihrem Feld.

2. Modellieren Sie ein übersichtliches und Ihnen gut bekanntes System (zur Not die Schulbibliothek)
 - a: Entwerfen Sie geeignete Klassen für die im Modell relevanten Objekte (z. B. Bücher, Autoren, Ausleihvorgänge, Mahnungen, ...). Benutzen Sie zur Beschreibung UML.

- b: Veranschaulichen Sie die im System erforderlichen Aktionen und ordnen Sie diese den Objekten zu.
- c: Beschreiben Sie die Abläufe in den Methoden durch Struktogramme.
- d: Erzeugen Sie einen **Prototyp**: Ein Programm, in dem die Methodenrumpfe noch weitgehend leer sind, in dem sich aber die Abläufe im System testen lassen.
- e: Füllen Sie die Methodenrumpfe mit Code.

1.2 Beispiel: Schiebepuzzle

Es erscheint mir hier nicht sinnvoll, alle Möglichkeiten zur Deklaration von Klassen (lokale, anonyme, abstrakte, ...) systematisch aufzuführen – dafür gibt es schließlich Bücher. Ich werde deshalb einige etwas umfangreichere Beispiele wählen, bei denen allerdings eine ganze Reihe unterschiedlicher Möglichkeiten benutzt wird. Zuerst ein Schiebepuzzle:



Das Spiel sollte wohl allen bekannt sein: Die Spielsteine sollen mithilfe des freien Platzes so verschoben werden, dass sie in steigender Folge geordnet sind.

1.2.1 Die Klasse der Spielsteine

Wir benötigen zuallererst Spielsteine. Die sollen die wesentlichen Daten der Steine (Farbe, Nummer, Position, ..) aufnehmen und sich auf einem Grafikkontext zeichnen können, der ihnen vom aufrufenden Programm übermittelt wird. Da alle Steine die gleiche Größe haben, definieren wir diese als *Klassenkonstante* namens *breite*. Sie existiert dann nur einmal und ist in allen Spielsteinen bekannt. Damit die Steine eine größere Schrift als normal nutzen, definieren wir einen geeigneten Font, auch als Klassenkonstante.

```
import java.awt.*;
```

```
public class Spielstein
{
```

```
    final static int breite = 50;
    final static Font font = new Font("Arial",Font.BOLD,20);
    int nummer;
    int x,y,b,h;
    Color hgf=Color.white;
```

Klassenkonstanten für die Größe und den Schrifttyp der Steine

diese Felder sind für jeden Stein anders, also werden Instanzvariablen vereinbart

Klassen benötigen Konstruktoren (im Notfall wird der geerbte benutzt). Wir wollen hier zwei Konstruktoren angeben, von denen der eine nur die absolut notwendige Information „Steinnummer“ als Parameter erhält, der andere zusätzlich die Farbe des Steins. Welcher dieser Konstruktoren aufgerufen wird, hängt von der Parameterliste des Aufrufs ab: die Konstruktoren werden *überladen*. Da der Code bei beiden fast identisch wäre, ruft der erste Konstruktor den zweiten mithilfe des Schlüsselworts *this* auf. (Dieser Aufruf muss als Erstes erfolgen!)

```
    public Spielstein(int n)
    {this(n,Color.white);}

    public Spielstein(int n, Color hgf)
    {
        nummer = n;
        this.hgf = hgf;
    }
```

überladene Konstruktoren

Jetzt benötigen wir nur noch einige Hilfsmethoden, um die Position und Farbe der Steine bei Bedarf zu verändern, und um sie „schön“ zu zeichnen. Der Grafikkontext *g* wird den Steinen (in unserem Fall) vom Fenster übergeben. In der *setPos*-Methode werden *x* und *y* als Parameter benutzt, liegen also im selben Namensraum wie die lokalen Variablen. Um die Instanzvariablen *x* und *y* ansprechen zu können (denn deren Werte sollen geändert werden), wird mit *this* auf diese zugegriffen.

```
    public void setPos(int x, int y) {
        this.x = x;
        this.y = y ;
    }
```

lokale Werte setzen unter Zugriff auf gleichnamige Instanzvariable

```
    public void setColor(Color c) {
        hgf = c;
    }
```

hier gibt es keine Namenskonflikte

```
    public void paint(Graphics g)
```

```
    {
        g.setColor(hgf);
        g.fillRect(x + 30, y + 50, breite, breite);
        g.setColor(Color.black);
        g.drawRect(x + 30, y + 50, breite, breite);
        if (nummer > 0)
        {
            g.setFont(font);
            g.drawString("" + nummer, x + 45, y + 83);
            g.setColor(Color.white);
            g.drawRect(x + 31, y + 51, breite - 2, breite - 2);
            g.setColor(Color.blue);
            for (int i = 1; i < 4; i++)
            {
                g.drawRect(x + 32, y + 52, breite - i - 2, breite - i - 2);
            }
        }
    }
```

Grafikkontext übergeben

Spielstein Nr. 0 nicht „ausmalen“

1.2.2 Das Fenster nutzt die Klasse der Spielsteine

Jetzt kommt das eigentliche Fenster, das auf Mausklicks reagieren soll. Es muss deshalb (z. B.) als *MouseListener* eingerichtet werden und die entsprechenden Methoden implementieren. Benötigt wird dafür die Bibliothek *java.awt.event*. Wir wollen die dafür erforderlichen Anweisungen zusammenfassen:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class Spiel extends Frame implements MouseListener
{
    /* ...

    public void init()
    {
        /*...
        addMouseListener(this);
        /*...

    }

    public void mouseClicked(MouseEvent e)
    {
        /*...
    }

    public void mousePressed(MouseEvent e)
    {
        /*...
    }

    public void mouseReleased(MouseEvent e)
    {
        /*...
    }

    public void mouseExited(MouseEvent e)
    {
        /*...
    }
}
```

Das Spiel selbst erfordert Instanzen von 24 Spielsteinen. Ein 25.ter wird als „freies Feld“ eingesetzt. Wir legen die einfach in einer Reihung ab.

```
Spielstein[] steine = new Spielstein[25];
```

In der *init*-Methoden werden diese Steine dann erzeugt.

```
for(int i=0;i<25;i++)
    if(i>0) steine[i] = new Spielstein(i,Color.green);
    else steine[i] = new Spielstein(i,Color.white);
```

Etwas komplizierter ist die Verwaltung des Spielfeldes. Wir wollen eine zweidimensionale Reihung benutzen, in deren Feldern wir einfach die aktuell dort befindlichen Steinnummern eintragen. `int[][] spielfeld = new int[5][5];`

Dazu müssen wir natürlich doppelt auftretende Zahlen verhindern. Wir benutzen eine Reihung mit booleschen Werten, in die wir schon gezogene Zahlen als *false* markieren. Zuerst müssen dann alle Werte auf *true* gesetzt werden.

```
boolean[] istNochFrei = new boolean[25];
for(int i=0;i<25;i++) istNochFrei[i] = true;
```

Danach können wir in der *init*-Methode solange Zahlen ziehen, bis wir eine neue gefunden haben. Wir benutzen eine Methode *zz*, die Zufallszahlen liefert.

```
private int zz(int von, int bis)
{
    return (int)Math.round(Math.random()*(bis-von)+von);
}

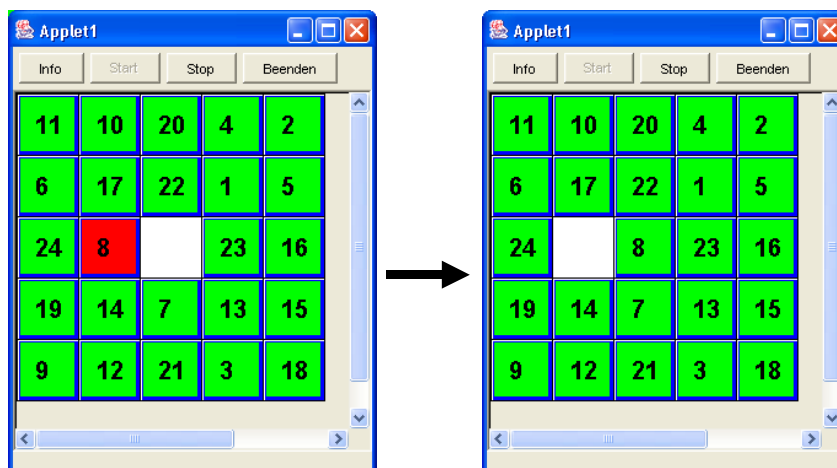
public void init()
{
    int n;
    /* ...
    for(int i=0;i<5;i++)
        for (int j=0;j<5;j++)
            {
                do n = zz(0,24);
                while(!istNochFrei[n]);
                istNochFrei[n] = false;
                spielfeld[i][j] = n;
                steine[n].setPos(50*i,50*j);
            }
    }
}
```

prüfen, ob die Zufallszahlen
schon mal gezogen wurden

Jetzt können wir alle Steine einmal zeichnen.

```
public void paint(Graphics g)
{
    for(int i=0;i<25;i++) steine[i].paint(g);
}
}
```

Das eigentliche Spielen geschieht, indem Mausaktionen ausgewertet werden. Der Einfachheit halber wählen wir das Ereignis *MouseReleased*. Wird die Maustaste über einem Spielstein das erste Mal losgelassen, dann wird der Stein rot eingefärbt. Wird die Taste danach über dem leeren Feld losgelassen und ist der Zug möglich (der Stein befindet sich also direkt über, unter oder neben dem leeren Feld), dann wird der leere Stein mit dem Spielstein vertauscht (der jetzt wieder grün dargestellt wird). Die neuen Positionen müssen auch im Spielfeld vermerkt werden. Eine boolesche Variable *ersterKlick* merkt sich, ob gerade ein Stein angeklickt oder verschoben werden soll.



```

public void mouseReleased(MouseEvent e)
{
    int x,y,n;

    if(ersterKlick)
    {
        x = (int)Math.round((e.getX()-30)/50);
        y = (int)Math.round((e.getY()-50)/50);

        if((x>=0) && (x<5) && (y>=0) && (y<5))
        {
            n = spielfeld[x][y];
            if (n>0)
            {
                steine[n].setColor(Color.red);
                steine[n].paint(getGraphics());
                nAlt = n;
                xAlt = x;
                yAlt = y;
                ersterKlick = false;
            }
        }
    }
    else
    {
        x = (int)Math.round((e.getX()-30)/50);
        y = (int)Math.round((e.getY()-50)/50);

        if((x>=0) && (x<5) && (y>=0) && (y<5))
        {
            n = spielfeld[x][y];
            if ((n==0) && (
                ((y==yAlt) && ((x==xAlt+1) || (x==xAlt-1))) ||
                ((x==xAlt) && ((y==yAlt+1) || (y==yAlt-1)))
            ))
            {
                spielfeld[xAlt][yAlt]= 0;
                spielfeld[x][y] = nAlt;
                steine[nAlt].setColor(Color.green);
                steine[nAlt].setPos(50*x,50*y);
                steine[nAlt].paint(getGraphics());
                steine[0].setPos(50*xAlt,50*yAlt);
                steine[0].paint(getGraphics());
            }
            else
            {
                steine[nAlt].setColor(Color.green);
                steine[nAlt].paint(getGraphics());
            }
        }
        ersterKlick = true;
    }
}

```

Koordinaten im Spielfeld aus den Mauskoordinaten bestimmen, die obere linke Ecke liegt bei (30|50)

Spielsteinnummer

Position und Nummer merken, Stein einfärben

... und beim nächsten Mal unten weitermachen!

s.o.

prüfen, ob der Zug möglich ist

Steine vertauschen und anzeigen

sonst: Fehlanzeige

Zur Übersicht noch einmal die Struktur des Gesamtprogramms:

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

public class FrameSchiebepuzzle extends Frame implements MouseListener
{

    public FrameSchiebepuzzle()
    {
        initComponents();
        setSize(300, 350);
        init();
    }

    private void initComponents() //wie angegeben
    public static void main(String args[]) //wie angegeben

    Spielstein[] steine = new Spielstein[25];
    int[][] spielfeld = new int[5][5];
    boolean ersterKlick = true;
    int nAlt,xAlt,yAlt;

    private int zz(int von, int bis) //wie angegeben

    public void init()
    {
        int n;
        setLayout(null);
        addMouseListener(this);

        for(int i=0;i<25;i++)
            if(i>0) steine[i] = new Spielstein(i,Color.green);
            else steine[i] = new Spielstein(i,Color.white);

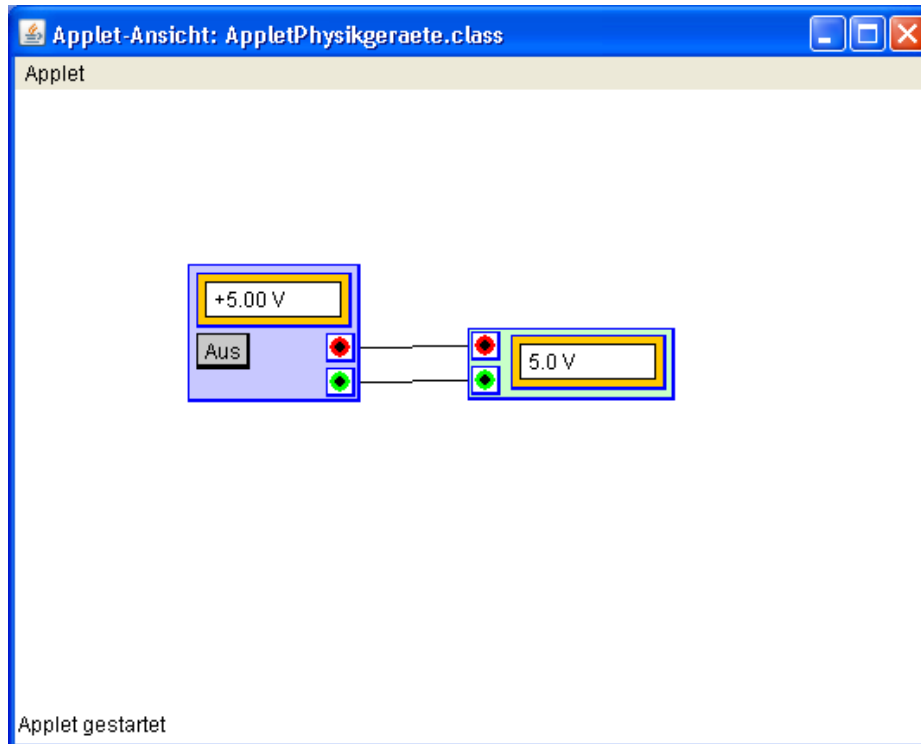
        boolean[] istNochFrei = new boolean[25];
        for(int i=0;i<25;i++) istNochFrei[i] = true;

        for(int i=0;i<5;i++)
            for (int j=0;j<5;j++)
            {
                do n = zz(0,24);
                while(!istNochFrei[n]);
                istNochFrei[n] = false;
                spielfeld[i][j] = n;
                steine[n].setPos(50*i,50*j);
            }
        ersterKlick = true;
    }

    public void paint(Graphics g) //wie angegeben
    public void mouseClicked(MouseEvent e) //wie angegeben
    public void mousePressed(MouseEvent e) //wie angegeben
    public void mouseReleased(MouseEvent e) //wie angegeben
    public void mouseExited(MouseEvent e) //wie angegeben
    public void mouseEntered(MouseEvent e) //wie angegeben
}
```

1.3 Beispiel: Physikobjekte

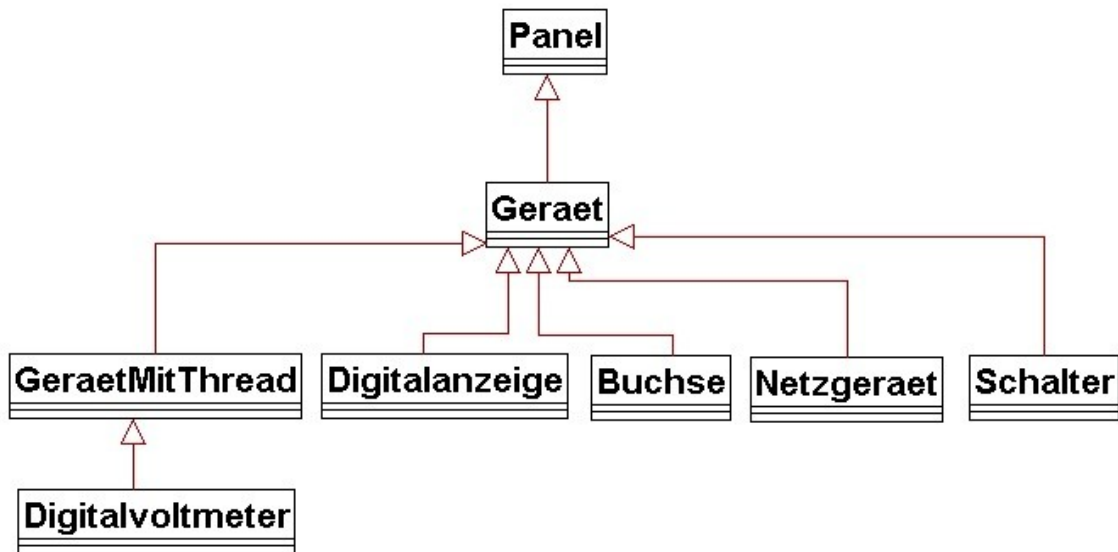
Als etwas komplexeres Beispiel für die unterschiedlichen Techniken, mit OOP umzugehen, wollen wir einen Satz von Physikgeräten entwickeln, die wir dann frei erzeugen und mit einander „verdrahten“ können. Ein einfaches Beispiel für einen „Versuch“ könnte dann so aussehen: Ein Festspannungs-Netzgerät wurde durch Mausklicks mit einem Digitalvoltmeter verbunden. Es zeigt jetzt den vom Netzgerät erzeugten Spannungswert an.



Betrachten wir diese und andere, für unsere Versuche geeignete Geräte, dann sehen wir schnell, welche immer wiederkehrenden Bauteile zur Konstruktion benötigt werden. Als kleine Auswahl nehmen wir:

- *Gehäuse*, die eine Position am Bildschirm, eine definierte Größe, unterschiedliche Farben für Oberfläche, ggf. die Beschriftung und den unglaublich effektvollen Rand haben (s.o.). Wir wollen diese Gehäuse als Mutterklasse aller Geräte, also als *Geraet* einführen.
- *Schalter*, mit denen Geräte eingestellt (z. B. eingeschaltet) werden können.
- *Buchsen*, über die Geräte miteinander verbunden werden können.
- *Anzeigen*, die z. B. Spannungswerte darstellen.
- *usw.*

Modellieren wir dieses System (in einer – aus Platzgründen – stark reduzierten Form), dann kommen wir zu einer Hierarchie von Klassen, die einerseits geeignete Bauteile beschreiben, andererseits fertige Geräte darstellen, die aus diesen Teilen zusammengesetzt werden. Weil die Darstellung „vor“ eventuellen Leitungen so besonders einfach wird, leiten wir unsere Geräte aus der *AWT-Panel*-Klasse ab, und weil wir später ziemlich autonom arbeitende Geräte benötigen, schieben wir noch eine Klasse *GeraetMitThread* ein.



Wie soll nun die Kommunikation zwischen den Geräten laufen?

1.3.1 Klassenvariable

Einerseits arbeiten die Geräte autonom, reagieren also auf Mausklicks, Mausbewegungen etc, andererseits müssen sie auch Informationen austauschen können und das Zeichnen z. B. von Verbindungslinien („Leitungen“) veranlassen, und sie müssen auch „arbeiten“. Wir müssen uns etwas einfallen lassen. Erfreulicherweise kennen wir schon einige der benötigten Techniken von anderen Beispielen her.

Unsere Geräte enthalten natürlich beschreibende Größen wie Position, Maße, Farben, ..., und sie implementieren das *MouseListener*-, ggf. noch das *MouseMotionListener*-Interface.

Beginnen wir mit den Mausereignissen. Einerseits sollen die Geräte am Bildschirm verschiebbar sein, andererseits sollen sie auf Mausklicks reagieren. Nun werden wir nicht die Buchsen auf einem Gerät verschieben, sondern das Gerät als Ganzes. Wir vereinbaren deshalb eine *boolesche Variable verschiebbar*, die definiert, ob das Gerät verschoben werden kann. Anfangs setzen wir ihren Wert auf *false*. Bei Bedarf kann er mithilfe der Methode *werdeVerschiebbar* geändert werden. Wenn wir schon mal dabei sind, dann vereinbaren wir auch noch eine *boolesche Variable sichtbar*, damit wir bei Bedarf das Gerät während des Verschiebens verstecken können.

Klicken wir auf ein verschiebbares Gerät, dann verstecken wir es und zeichnen stattdessen ein einfaches Rechteck. Das gehört aber auf die Zeichenfläche des Applets, und die muss dem Gerät irgendwie zugänglich gemacht werden. Eine Möglichkeit dafür ist die Vereinbarung einer *Klassenvariable Zeichenblatt*, die vom Applet gesetzt werden muss und auf der die Geräte „zeichnen“. Die Geräteklasse vereinbart dann

```
static Graphics zeichenblatt;
```

Klassenvariable

Das Applet belegt diese mit einem Wert – unabhängig vom konkreten Gerät

```
Geraet.zeichenblatt = getGraphics();
```

und das Gerät zeichnet darauf

```
zeichenblatt.drawRect(this.x,this.y,b,h);
```

Später können wir die Leitungen zwischen den Gerätebuchsen auf die gleiche Art erzeugen.

1.3.2 Verschiebbare Geräte

Wir beginnen jetzt also mit dem Verschieben, wenn eine Maustaste gedrückt wurde – allerdings nur, wenn das Gerät auch verschiebbar ist!

```
public void mousePressed(MouseEvent e)
{
    if(verschiebbar)
    {
        int x = e.getX(), y = e.getY();
        xOffset = x-this.x;
        yOffset = y-this.y;
        setVisible(false);
        zeichenblatt.setXORMode(Color.white);
        zeichenblatt.setColor(Color.black);
        sichtbar = false;
    }
}
```

Stelle merken ...

... und die Position des
Mausklicks relativ zur
Geräteposition auch ...

... und in den
invertierenden
Modus schalten.

Danach wird bei gedrückter Maustaste die neue, verschobene Position durch ein Rechteck angezeigt.

```
public void mouseDragged(MouseEvent e)
{
    if(verschiebbar)
    {
        int x = e.getX(), y = e.getY();
        if(sichtbar)
        {
            zeichenblatt.drawRect(this.x,this.y,b,h);
            sichtbar = false;
        }
        else
        {
            this.x = x - xOffset;
            this.y = y - yOffset;
            zeichenblatt.drawRect(this.x,this.y,b,h);
            sichtbar = true;
        }
    }
}
```

Rechteck löschen
(durch Nochmal-Zeichnen)

sonst Rechteck zeichnen

Zuletzt – nach dem Loslassen der Maustaste – wird das Gerät an der neuen Position dargestellt.

```
public void mouseReleased(MouseEvent e)
{
    int x = e.getX(), y = e.getY();
    if(verschiebbar)
    {
        if(sichtbar) zeichenblatt.drawRect(this.x,this.y,b,h);
        this.x = x - xOffset;
        this.y = y - yOffset;
        setBounds(this.x,this.y,b,h);
        zeichenblatt.setPaintMode();
        setVisible(true);
    }
    else verarbeite(e.getX(),e.getY());
}
```

Rechteck ggf. löschen und
Gerät zeigen

Ausweichverfahren für nicht
verschiebbare Geräte

1.3.3 Virtuelle Methoden und Polymorphismus

Wenn Geräte nicht verschiebbar sind, dann müssen sie vielleicht anders auf Mausklicks reagieren. Leider wissen wir noch gar nicht, um was für ein Gerät es sich später handeln wird und wie dieses zu reagieren hat. Wir greifen deshalb auf Javas Eigenschaft zurück, ausschließlich *virtuelle Methoden* zu implementieren.

Wenn eine Klasse über eine Methode verfügt, dann kann eine Tochterklasse diese *überlagern*, d. h. durch eine gleichnamige ersetzen. Verschiedene Klassen können also unter dem gleichen Namen verschiedenes – sinnvoller Weise meist aber äquivalentes – Verhalten implementieren. Man spricht von *Polymorphismus*. Java entscheidet erst zum Zeitpunkt des Aufrufs einer Methode darüber, welche der möglichen die richtige ist, und diese wird dann ausgeführt. Man spricht von *dynamischer Bindung*. Javaklassen verfügen über eine Tabelle, in der die Adressen der für diese Klasse verfügbaren Methoden aufgeführt sind: die *virtuellen Methodentabelle* VMT. Wird eine Methode von einer anderen aufgerufen, dann muss diese natürlich vorhanden sein – z. B. als leerer Rumpf¹. Sie kann später von Tochterklassen durch eine andere ersetzt werden. Obwohl zum Zeitpunkt der Compilierung der einen Methode die andere noch leer war, wird zur Zeit der Ausführung die richtige gewählt.

Genau diesen Mechanismus nutzen wir jetzt aus:

Für alle Geräte vereinbaren wir eine leere Methode

```
public void verarbeite(int x, int y)
{
}
```

Beim Loslassen der Maustaste wird diese für nicht verschiebbare Objekte mit den Koordinaten des Mauszeigers aufgerufen.

```
... else verarbeite(e.getX(),e.getY());
```

Normalerweise geschieht dann nichts. Überlagern wir die Methode aber in einer Tochterklasse, z. B. in der Klasse **Schalter**

```
public void verarbeite(int x, int y)
{
    if(zustand==0) zustand = 1;
    else zustand = 0;
    repaint();
    ...
}
```

dann wird diese ausgeführt.

¹ alternativ könnte man auch mit abstrakten Methoden arbeiten

1.3.4 Kommunikation zwischen Objekten

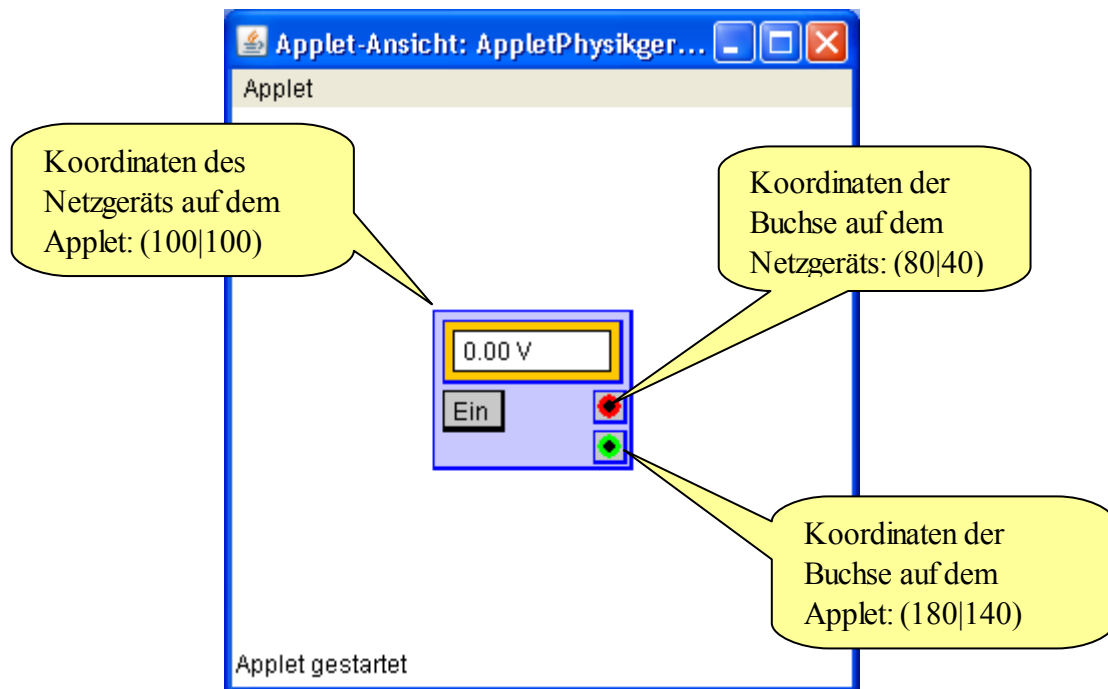
Wird eine Buchse mit der Maus angeklickt, dann könnte das der Anfang einer Aktionskette sein, die zu einer Verbindung zwischen zwei Geräten führt. Wie erfährt aber das andere Gerät von dieser Buchse, und wo erhalten wir die Koordinaten der Linien her, die die Leitungen auf dem Bildschirm darstellen sollen?

Das erste Problem lösen wir wieder durch *Klassenvariable*: Wir führen zwei Buchsen ein, denen wir bei Bedarf den Wert interner Buchsen zuweisen. (Da es sich um Objekte handelt, werden also die Referenzen auf diese Buchsen gespeichert.)

```
static Buchse Eingangsbuchse=null, Ausgangsbuchse=null;
```

Als Klassenvariable stehen diese Buchsen dann allen Objekten zur Verfügung.

Das Koordinatenproblem lösen wir anders: Jedes Gerät kennt „seine“ Koordinaten innerhalb eines anderen Objekts. Die Position auf der Zeichenfläche ergibt sich dann als Kombination solcher Gerätepositionen.



Wir versehen deshalb alle Geräte mit einer Referenz auf ihren „Besitzer“, also auf das Gerät, auf dem sie sich befinden. Diese Information können wir dann bei Bedarf nutzen.

1.3.5 Überladene Konstruktoren

Geräte werden wie alle Objekte mithilfe von *Konstruktoren* erzeugt. Normalerweise werden diesen Konstruktoren Informationen über das zu erzeugende Objekt mit übergeben, z. B. seine Position, seine Farbe, ... Es kann nun durchaus sinnvoll sein, verschiedene Konstruktoren einzuführen, die mehr oder weniger Zusatzinformationen beinhalten: Die Konstruktoren werden *überladen*. Sie haben den gleichen Namen (wie die Klasse), aber unterschiedliche Parameterlisten. Anhand dieser werden sie dann auch unterschieden.

Meist ruft man innerhalb eines überladenen Konstruktors den mit der umfangreichsten Parameterliste als Erstes auf. Dabei ersetzt man die fehlenden Werte durch *Defaultwerte*. Wir nutzen diese Technik auch für die *Geraete*-Klasse. Übergeben wird mindestens der Besitzer.

```
public Geraet(Geraet owner)
{
    this(100,100,80,50,Color.white,owner);
}

public Geraet(int x, int y, int b, int h, Geraet owner)
{
    this(x,y,b,h,Color.white,owner);
}

public Geraet(int x, int y, int b, int h, Color hgf, Geraet owner)
{
    this.x = x;
    this.y = y;
    this.b = b;
    this.h = h;
    setLayout(null);
    setBounds(x,y,b,h);
    setBackground(hgf);
    setForeground(sf);
    besitzer = owner;
    addMouseListener(this);
}
```

den anderen Konstruktor mit übergebenen und Defaultwerten aufrufen

ebenso

das isser!

1.3.6 Die Klasse der Geraete

Zur Übersicht noch mal die gesamte Klasse – mit einigen Hilfsmethoden.

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

public class Geraet extends Panel
    implements MouseMotionListener, MouseListener
{
    int x,y,b,h;
    Color hgf=Color.white,rf=Color.blue,sf=Color.black;
    Geraet besitzer;
    static Graphics zeichenblatt;
    int xOffset,yOffset;
    boolean sichtbar,verschiebbar=false;
    static Buchse Eingangsbuchse=null, Ausgangsbuchse=null;

    public void setColor(Color c)
    {hgf = c;}
}
```

Farbe ändern

```
public Geraet(Geraet owner) //default-Konstruktor
{this(100,100,80,50,Color.white,owner);}

public Geraet(int x, int y, int b, int h, Geraet owner)
{this(x,y,b,h,Color.white,owner);}

public Geraet(int x, int y, int b, int h, Color hgf, Geraet owner)
{
    this.x = x; this.y = y; this.b = b; this.h = h;
    setLayout(null); setBounds(x,y,b,h);
    setBackground(hgf); setForeground(sf);
    besitzer = owner;
    addMouseListener(this);
}
public void werdeVerschiebbar()
{
    verschiebbar = true; addMouseMotionListener(this);
}

public void verarbeite(int x, int y){}

public void paint(Graphics g)
{
    g.setColor(rf);          g.drawRect(0,0,b-1,h-1);
    g.drawLine(0,h-2,b-1,h-2); g.drawLine(b-2,0,b-2,h-1);
}

public void mousePressed(MouseEvent e)
{
    if(verschiebbar)
    {
        int x = e.getX(), y = e.getY();
        xOffset = x-this.x; yOffset = y-this.y;
        setVisible(false); sichtbar = false;
        zeichenblatt.setXORMode(Color.white);
        zeichenblatt.setColor(Color.black);
    }
}

public void mouseDragged(MouseEvent e)
{
    if(verschiebbar)
    {
        int x = e.getX(), y = e.getY();
        if(sichtbar)
        {zeichenblatt.drawRect(this.x,this.y,b,h); sichtbar = false;}
        else
        {
            this.x = x - xOffset; this.y = y - yOffset;
            zeichenblatt.drawRect(this.x,this.y,b,h);
            sichtbar = true;
        }
    }
}

public void mouseReleased(MouseEvent e)
{
    int x = e.getX(), y = e.getY();
    if(verschiebbar)
    {
        if(sichtbar) zeichenblatt.drawRect(this.x,this.y,b,h);
    }
}
```

Geräte
verschiebbar

Geräte mit 3D-
Rand zeichnen

```

        this.x = x - xOffset;
        this.y = y - yOffset;
        setBounds(this.x, this.y, b, h);
        zeichenblatt.setPaintMode();
        setVisible(true);
    }
    else verarbeite(e.getX(), e.getY());
}

public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mouseMoved(MouseEvent e) {}
}

```

1.3.7 Die Klasse der GeraeteMitThread

Einige Geräte müssen unabhängig von Benutzeraktionen arbeiten, z. B. laufend die „Spannungen“ an den Buchsen überprüfen und darstellen. Ansonsten unterscheiden sie sich nicht von anderen Geräten. Wir implementieren also zusätzlich das Interface *Runnable*, schreiben eine *run()*-Methode und starten einen *Thread*, der die Methode laufend aufruft. Um den Thread geordnet beenden zu können, fügen wir eine *stop()*-Methode dazu. Wie bei den Mausklicks ist auch hier zu diesem Zeitpunkt nicht klar, welche Aktion vom Thread ausgelöst werden soll. Wir rufen deshalb die leere Methode *arbeite()* auf, die von den Tochterklassen überlagert wird.

```

import java.awt.*;
import java.awt.event.*;

public class GeraetMitThread extends Geraet implements Runnable
{
    Thread t;
    public void stop()
    {
        t.interrupt();
    }

    public GeraetMitThread(Geraet owner)
    {
        this(100, 100, 80, 50, Color.white, owner);
    }

    public GeraetMitThread(int x, int y, int b, int h, Geraet owner)
    {
        this(x, y, b, h, Color.white, owner);
    }

    public GeraetMitThread(int x, int y, int b, int h, Color hgf, Geraet owner)
    {
        super(x, y, b, h, hgf, owner);
        t = new Thread(this);
        t.start();
    }

    public void arbeite()
    {
    }
}

```

Thread vereinbaren, ...

..., ggf. stoppen, ...

..., starten ...

```
public void run()
{
    while(true)
    {
        if(t.isInterrupted()) break;
        try
        {
            t.sleep(10);
            arbeite();
        }
        catch(InterruptedException e){}
    }
}
}
```

... und nutzen.

1.3.8 Die Klasse der Digitalanzeigen

Digitalanzeigen sollten über eine Reihe überladener Konstruktoren verfügen, mit denen sie auf vielfältige Weise erzeugt werden können. Ansonsten kann ihre Aufschrift geändert werden, und zeichnen müssen sie sich natürlich können.

```
import java.awt.*;

public class DAnzeige extends Geraet
{
    String aufschrift="";

    public DAnzeige(Geraet owner)
    {super(owner);}

    public DAnzeige(int x,int y, int b, int h, Geraet owner)
    {super(x,y,b,h,owner);}

    public DAnzeige(int x,int y, int b, int h, String s, Geraet owner)
    {super(x,y,b,h,owner); aufschrift = s;}

    public DAnzeige(int x,int y, int b, int h, Color hgf, Geraet owner)
    {super(x,y,b,h,hgf,owner);}

    public DAnzeige(int x,int y,int b,int h,Color hgf,String s,Geraet owner)
    {super(x,y,b,h,hgf,owner); aufschrift = s;}

    public void setText(String s)
    {
        aufschrift = s;
    }

    public void paint(Graphics g)
    {
        super.paint(g);
        g.setColor(Color.white);
        g.fillRect(5,5,b-12,h-12);
        g.setColor(Color.black);
        g.drawRect(5,5,b-12,h-12);
        g.drawString(aufschrift,10,20);
    }
}
}
```

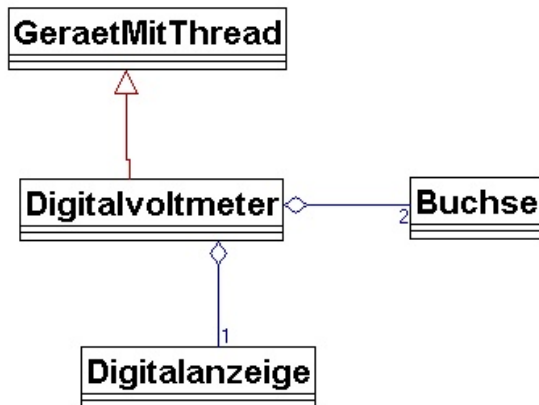
Konstruktoren

Anzeigestring ändern ...

... und darstellen

1.3.9 Die Klasse der Digitalvoltmeter

Digitalvoltmeter verfügen über eine Digitalanzeige und zwei Buchsen. Da sie kontinuierlich arbeiten, sind sie Abkömmlinge der *GeraeteMitThread*-Klasse. Entsprechend überlagern sie die *arbeite*-Methode durch eine eigene, in der die Spannungswerte an den Buchsen abgefragt werden.



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

```
public class DVoltmeter extends GeraeteMitThread
{
```

```
    DAnzeige anzeige;
    Buchse eingang, ausgang;
    double alterWert = 0;
```

```
    public DVoltmeter(int x,int y)
```

```
    {
        super(x,y,120,42,new Color(200,255,200),null);
        anzeige = new DAnzeige(25,4,90,32,Color.orange,"0.00 V",this);
        add(anzeige);
        eingang = new Buchse(2,2,Color.red,"Eingang",this);
        add(eingang);
        ausgang = new Buchse(2,22,Color.green,"Ausgang",this);
        add(ausgang);
    }
```

Voltmeter mit
Digitalanzeige und zwei
Buchsen erzeugen

```
    public void arbeite()
```

```
    {
        double neuerWert = eingang.wert()-ausgang.wert();
        if(!(alterWert==neuerWert))
        {
            anzeige.setText(""+neuerWert+" V");
            alterWert = neuerWert;
            anzeige.repaint();
        }
    }
}
```

Spannungswerte
bestimmen und anzeigen
(nur bei Änderungen, weil
es sonst so flimmert)

1.3.10 Die Klasse der Buchsen

Die interessantesten Bauteile sind zweifellos die Buchsen, denn erst über diese interagieren die Geräte. Der Einfachheit halber unterscheiden wir Ein- und Ausgänge. Ein *Ausgang* hat einen *Wert*, der von der Funktionsweise des Gerätes bestimmt wird, zu dem die Buchse gehört. *Eingänge* sollen eine Referenz *kontakt* besitzen, die auf den Ausgang zeigt, mit dem sie verbunden wurden. Bei unverbundenen Eingängen hat die Referenz den Wert *null*.

```
import java.awt.*;

public class Buchse extends Geraet
{
    Buchse kontakt=null;
    double wert=0;
    String typ="Ausgang";
```

Unverbundene Eingänge liefern den Wert 0, verbundene den Wert des Ausgangs, mit dem sie verbunden wurden.

```
public double wert()
{
    if(typ.equals("Eingang"))
        if(kontakt==null)
            return 0;
        else
            return kontakt.wert();
    else return wert;
}
```

Der Buchsen-Konstruktor legt neben den üblichen Werten auch fest, um welchen Typ von Buchse es sich handeln soll.

```
public Buchse(int x,int y, Color c, String t, Geraet owner)
{
    super(x,y,17,17,new Color(200,200,200),owner);
    sf = c;
    if(t.equals("Eingang")) typ = "Eingang"; else typ = "Ausgang";
}
```

Jetzt müssen die Buchsen noch verbunden werden können. Das geschieht auf Mausclicks, die in unserem Modell die Methode *verarbeite* aufrufen. Hier wird unterschieden, um welchen Buchsentyp es sich handelt, und die Kommunikation läuft über die Klassenvariablen *Eingangsbuchse* und *Ausgangsbuchse*, die normalerweise den Wert *null* haben. Liegt die Partnerbuchse noch nicht fest, dann vermerken wir die Adresse der angeklickten Buchse als Wert in der entsprechenden Klassenvariablen, z. B. als

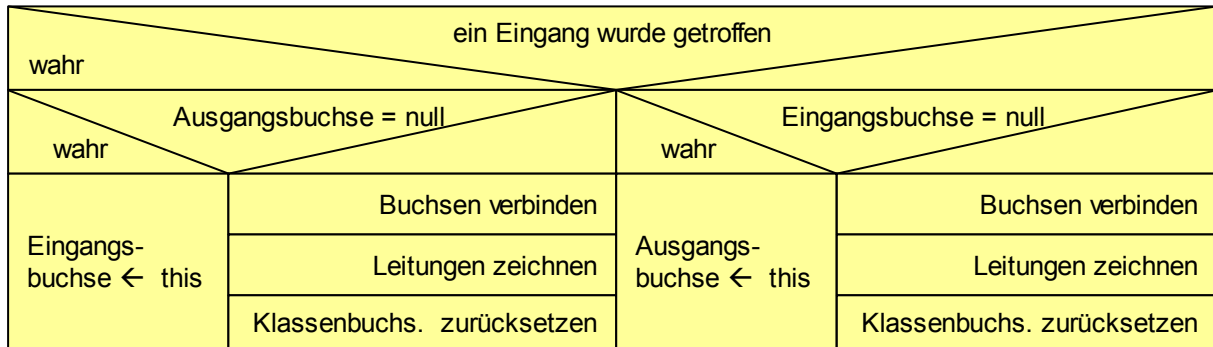
```
Eingangsbuchse = this;
```

Haben aber beide Klassenvariablen vernünftige Werte, dann werden die Buchsen verbunden, z. B. über.

```
setzeVerbindung(Ausgangsbuchse);
```

Außerdem werden die Leitungen gezeichnet. Zusätzlich werden angeklickte Buchsen farblich hervorgehoben.

public void verarbeite(int x int y)



```

public void verarbeite(int x, int y)
{
    int xanf=0, yanf=0, xend=0, yend=0;
    if (typ.equals("Eingang"))
    {
        if (Ausgangsbuchse != null)
        {
            setzeVerbindung (Ausgangsbuchse);
            Ausgangsbuchse.setBackground (Ausgangsbuchse.hgf);
            Ausgangsbuchse.repaint ();
            setBackground (hgf); repaint ();
            zeichenblatt.setPaintMode (); zeichenblatt.setColor (Color.black);
            xanf = Ausgangsbuchse.besitzer.x+Ausgangsbuchse.x+8;
            yanf = Ausgangsbuchse.besitzer.y+Ausgangsbuchse.y+8;
            xend = besitzer.x+this.x+8;
            yend = besitzer.y+this.y+8;
            zeichenblatt.drawLine (xanf, yanf, xend, yend);
            Ausgangsbuchse = null; Eingangsbuchse = null;
        }
        else
        {
            Eingangsbuchse = this;
            setBackground (Color.blue); repaint ();
        }
    }
    else
    {
        if (Eingangsbuchse != null)
        {
            Eingangsbuchse.setzeVerbindung (this);
            Eingangsbuchse.setBackground (Eingangsbuchse.hgf);
            Eingangsbuchse.repaint ();
            setBackground (hgf); repaint ();
            zeichenblatt.setPaintMode (); zeichenblatt.setColor (Color.black);
            xanf = besitzer.x+this.x+8;
            yanf = besitzer.y+this.y+8;
            xend = Eingangsbuchse.besitzer.x+Eingangsbuchse.x+8;
            yend = Eingangsbuchse.besitzer.y+Eingangsbuchse.y+8;
            zeichenblatt.drawLine (xanf, yanf, xend, yend);
            Ausgangsbuchse = null; Eingangsbuchse = null;
        }
        else {
            Ausgangsbuchse = this;
            setBackground (Color.blue); repaint ();
        }
    }
}

```

Buchsen verbinden und Leitungen zeichnen

es war nur der erste Teil der Verbindung

Buchsen verbinden und Leitungen zeichnen

es war nur der erste Teil der Verbindung

Zur Übersicht noch mal die gesamte Klasse.

```
import java.awt.*;

public class Buchse extends Geraet
{
    Buchse kontakt=null;
    double wert=0;
    String typ="Ausgang";

    public Buchse(int x,int y, Color c, String t, Geraet owner) //wie angegeben

    public void paint(Graphics g)
    {
        super.paint(g);
        g.setColor(sf);
        g.fillOval(2,2,12,12);
        g.setColor(Color.black);
        g.fillOval(5,5,6,6);
    }

    public void setzeVerbindung(Buchse b)
    {
        if(typ.equals("Eingang"))
            kontakt = b;
        else kontakt = null;
    }

    public double wert()//wie angegeben

    public void verarbeite(int x, int y) //wie angegeben
}

```

Buchsen zeichnen

Buchsen ggf. verbinden

1.3.11 Aufgaben

Die Physikgeräte ermöglichen eine Fülle sehr unterschiedlicher Aufgabenstellungen, die

- für Gruppenarbeit sehr geeignet sind („Jede Gruppe baut und testet ihr eigenes Gerät.“). Die neu erzeugten Klassen können dann in gemeinschaftlichen Versuchen eingesetzt werden. Dafür sind Absprachen z. B. über die Zusammenarbeit der Klassen wichtig – sonst funktioniert es einfach nicht.
- zur Leistungsdifferenzierung dienen können („Es gibt halt unterschiedlich komplizierte Geräte.“) Bauen Sie mal einen x-y-t-Schreiber!
- neben einfachen Aufgaben („neue Geräte nur aus Bauteilen zusammensetzen“) auch sehr schwierige ermöglichen („automatisches Leitungen verlegen: Autorouting“ oder „Überprüfen der Schaltung auf Kurzschlüsse“).

Hier eine kleine Auswahl:

1. Bauen Sie neue **Spannungsquellen**:
 - a: eine Gleichspannungsquelle mit veränderbarer Ausgangsspannung.
 - b: eine Wechselspannungsquelle.
 - c: einen Funktionsgenerator, der z. B. Dreiecks- oder Sägezahnspannungen erzeugen kann.
2. Bauen Sie neue **Anzeigegeräte**:
 - a: ein Analogvoltmeter, das Spannungen mithilfe eines Zeigers und einer Skala anzeigt.
 - b: ein Oszilloskop, das insbesondere Wechselspannungssignale zeigt.
 - c: einen Schreiber, der den zeitlichen Verlauf von Spannungen aufzeichnet.
 - d: eine Glühbirne (LED), die kaputt geht, wenn die anliegende Spannung zu hoch wird.
3. Entwickeln Sie eine Klasse **Leitung** zur Verbindung von Buchsen, deren Verlauf am Bildschirm durch Mausklicks gesetzt/verändert werden kann. Diskutieren Sie das Problem gründlich, es ist nicht ganz einfach.
4. Entwickeln Sie einen Versuchsaufbau zum Test von Bauteilen in einem **einfachen Stromkreis**:
 - a: Diskutieren Sie dazu das Problem der Strommessung.
 - b: Entwickeln Sie unterschiedliche Bauteile: Widerstände, Kondensatoren, Dioden, Spulen, ...
 - c: Führen Sie eine Möglichkeit ein, die Bauteile zu wechseln und die Ergebnisse auszugeben.
 - d: Protokollieren Sie die Versuchsergebnisse durch geeignete Aufzeichnungsgeräte, z. B.: indem die Werte in eine Tabelle eingetragen, ggf. grafisch dargestellt werden.