

1.	Warum Java ? - eine knappe Erläuterung	2
2.	Ein erstes interaktives Programm	4
2.1	Standardklassen für Applets.....	5
	Buttons und Labels.....	6
	Verschiedene Appletviewer	6
	Farben und Fonts	8
	Beispielprogramme	9
	Übungen.....	10
2.2	Interaktives Ballspielen.....	12
2.3	Layoutmanager	16
	Beispiel	19
	Panels.....	20
	TextField, TextArea, Canvas, Choice.....	21
	Beispiel	24
	Ballspielen (Version 2)	26
	Aufgaben.....	30
2.4	Variationen zum ActionListener.....	31
3.	Eine eigene Klasse und Mausereignisse	33
	Klassendiagramm	33
	MouseListener, MouseMotionListener.....	35
	Beispielprogramm	36

1. Warum Java ?

"Java ist cool. Java wollen alle. Java gehört die Zukunft. Java kann alles."

Das konnte man im Laufe der letzten 30 Jahre von jeder Computersprache hören. Es läßt sich trefflich streiten, welches denn nun die richtige Computersprache für den Informatikunterricht - und für alle andern Zwecke - sein soll. Wenn es nur darum ginge, Programmieren zu lernen oder eine Computersprachekurs abzuhalten, dann wäre der Streit wohl angebracht. Da das aber nicht im Mittelpunkt unseres Interesses steht, reichen hier ein paar Informationen aus.

Jeder Anfänger muss mindestens einmal in seinem Programmiererleben das berühmte "Hello World" programmiert haben.

BASIC (1977 - 1982)

```
10 PRINT "Hello World"
```

PASCAL (1980 - heute)

```
program Hallo;
Begin
  writeln('Hello World');
End.
```

C / C++ (1990 - heute)

```
#include <stdio.h>

int main()
{
  printf("\n Hello World \n ");
  return 0;
}
```

JAVA(cool)

HTML - Datei

```
<HTML>
<BODY>
<APPLET code = "Hello.class" width = 300 height = 400>
</APPLET>
</BODY>
</HTML>
```

Die allein reicht aber nicht und hat mit Java auch nichts zu tun

JAVA-Datei

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

public class Hello extends Applet
{
public void init()
{
System.out.println("Hello World");
}
}
```

Das funktioniert aber nur, wenn der Browser, in dem das Applet abläuft, eine sogenannte Console als Ausgabefenster bereitstellt. Wenn das nicht der Fall ist, muss es möglicherweise etwas anders aussehen:

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

public class Hello extends Applet
{
public void init()
{
repaint();
}
public void paint( Graphics g )
{
g.drawString("Hello World");
}
}
```

Dieses Beispiel ist unfair und nicht nett, denn es vermittelt den Eindruck, dass doch wohl Basic die einfachste und effektivste Sprache sei. Das ist sicher nicht der Fall. Es wird aber deutlich, dass nicht immer die aktuellste Sprache für jeden Zweck die beste sein muss.

Der große Vorteil von Java ist die "Plattformunabhängigkeit". Dieses Schlagwort soll bedeuten, dass Javaprogramme auf einem beliebigen (geeigneten) Computer geschrieben werden können und dann auf jedem anderen (geeigneten) Computer mit irgendeinem anderen Betriebssystem ablaufen können.

Außerdem ist man das Problem los, dass jedes Betriebssystem eigene Grafikbefehle und andere Spezialitäten bereithält, die eine Übertragung von Programmtexten von einem zum anderen System unmöglich macht. Es war zu Turbopascalzeiten ja sogar so schlimm, dass sich die Grafikbefehle von Version zu Version änderten und plötzlich alte Programme nicht mehr funktionierten.

Das hat mit Java ein Ende und daher bietet es sich an Java auch im Unterricht zu betreiben.

2. Ein erstes interaktives Programm

Nachdem nun in den ersten Übungen im vorangegangenen Abschnitt ein Ball im Appletfenster hin - und her fliegt, soll es nun interaktiv werden (wieder so ein Schlagwort !). Interaktiv sind Programme schon immer gewesen. Auch Basicprogramme aus dem Jahre 1979 bewirkten eine Interaktion mit dem Benutzer. Wenn man aufgefordert wurde, einen Wert einzugeben, dann war das sicher schon Interaktion. Gemeint ist heute aber, dass sich der Anwender aussuchen kann, was er als Nächstes tun möchte. Man kann Buttons mit der Maus anklicken (wenn es welche gibt) oder aus Klappmenüs Punkte auswählen oder in ein geeignetes Textfeld irgend etwas eintragen oder das Programm "mittendrin " verlassen. Das alles war in sequentiellen Programmen nicht möglich. Da wurde vom Programm vorgeschrieben, was wann zu tun ist. (Wir alle wünschen uns nun interaktive Behörden und Ämter).

Damit sowas klappt, lauert bei der Abarbeitung interaktiver Programme ein *EventManager* im Hintergrund, der andauernd nachschaut, ob der Anwender irgend ein *Event* auslöst (Event= Ereignis und Geburtstagsfeier = MegaEvent). Dabei gibt es Tastaturevents, Mausevents und bestimmt noch viele andere, die uns im Moment nicht so sehr interessieren. Der Eventmanager fragt also andauernd in einer unendlichen Schleife alle Events ab. Tut die Benutzerin / der Benutzer nichts, dann hat der Manager auch nichts zu tun.

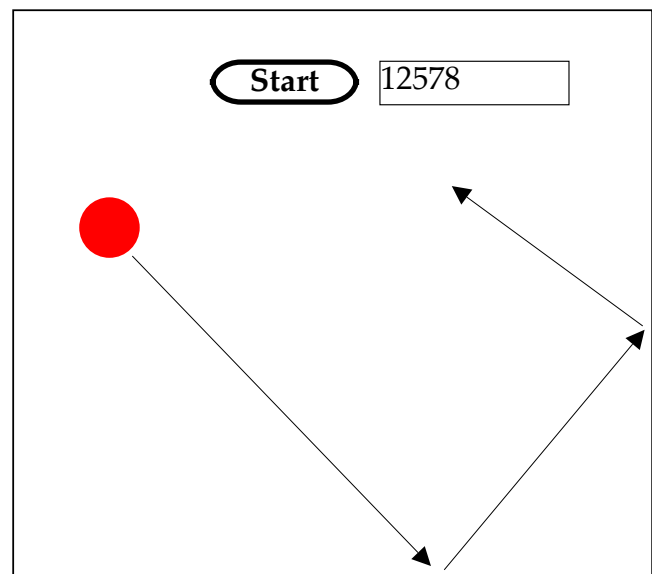
Wir wollen nun in einem ersten kleinen Beispiel ein Mausevent erzeugen, in dem wir auf einen Button klicken (WOW !). So ein Button oder Knopf wird neudeutsch auch "Schaltfläche" genannt, was bestimmt die schlechteste Bezeichnung ist, die man sich vorstellen kann.

Ziel der Übung:

Das Ballprogramm soll erweitert werden. Auf der Appletfläche soll ein Button erscheinen, der als Beschriftung "Start" enthält. Was der wohl tun soll, wird hier noch nicht verraten. Außerdem soll im Appletfenster ein kleines Textfeld erscheinen, das uns anzeigt, wie oft der Ball schon an die Ränder angestoßen ist.

Das könnte dann wie nebenstehend aussehen.

zunächst müssen wir uns mit den notwendigen Standardklassen beschäftigen, die wir für dieses kleine Projekt brauchen.



Hinweis:

Bei den folgenden Beispielen ist der HTML-Code immer gleich. Er wurde in den Anfangsbeispielen schon vorgestellt.

2. 1. Standardklassen für Applets

Für nette Bildschirmdarstellungen haben wir es mit einer Reihe von Standardklassen zu tun, aus denen wir unsere Objekte ableiten.

Schaltflächen oder auch Knopf oder auch Button abgeleitet aus `java.awt.Component`

```
Deklaration:      Button    meinKnopf;
                  .....
                  meinKnopf = new Button (" Hallo ");
```

Es wird ein Button erzeugt, der den Text "Hallo" trägt. Man kann ihn schon mit der Maus bedienen, es passiert aber erstmal nichts.

Zur Klasse Button gehören einige Methoden:

```
setLabel( String s)    beschriftet den Button mit dem String s, wenn der nicht schon
                       beschriftet war.
```

```
Beispiel:  meinKnopf= new Button();
           meinKnopf.setLabel("Auweia");
```

Textflächen, die nicht bearbeitet werden können (Labels) aus `java.awt.Component`

```
Deklaration:      Label    meinLabel;
                  .....
                  meinLabel= new Label (" Hallo ");
```

Hier wird ein Label erzeugt, das auch gleich die Beschriftung "Hallo" bekommt.

Zur Klasse Label gehören noch ein paar Methoden:

```
setText( String s)      beschriftet das Label mit dem String s
setBackground(Color c) setzt eine Hintergrundfarbe für das Label mit der Farbe c
setForeground(Color c)  setzt eine Schriftfarbe für das Label mit der Farbe c
```

Dazu nun ersteinmal ein kleines Programm:

```
/*Ein erstes Programm mit einem Button und einem Textlabel    */
/* VLIN 11/2001                                              */

import java.awt.*;      // standard Import für application window tools
import java.applet.Applet; // wenn auf dieses Zeile verzichtet wird, muss es bei
                          // der Klassendefinition am Ende ... java.applet.Applet sein

public class test3 extends Applet
{
    Button meinButton1; // eine Deklaration
    Label  meinLabel1;  // nun die Beschriftungen neben den Knöpfen
```

```

public void init() {
    meinButton1 = new Button("Start");
    meinLabel1 = new Label(" 0 Boing");
    add(meinButton1);
    add(meinLabel1);
}
}

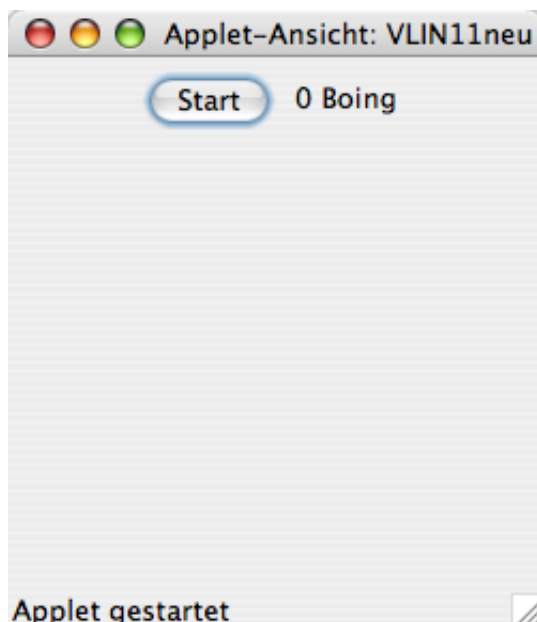
```

// Beginn von init

// new Button ist Standardroutine
// am Anfang auf Null gesetzt
// füge dem Appletwindow etwas hinzu
// und noch was

// fertig mit Applet

Probiert man dieses kleine Programm aus, erhält man ein Applet, das noch nicht ganz so ist, wie wir es uns wünschen.



Applet mit dem sun Applet Viewer



Applet mit dem Internet Explorer

Verschiedene Applet-Anschau-Programme und Internetbrowser zeigen das Ergebnis verschieden an. Mal ist der Button im 3D-Look zu sehen, mal nicht. Dann ist der Knopf mal mit weißem Hintergrund und mal mit grauem Hintergrund zu sehen.

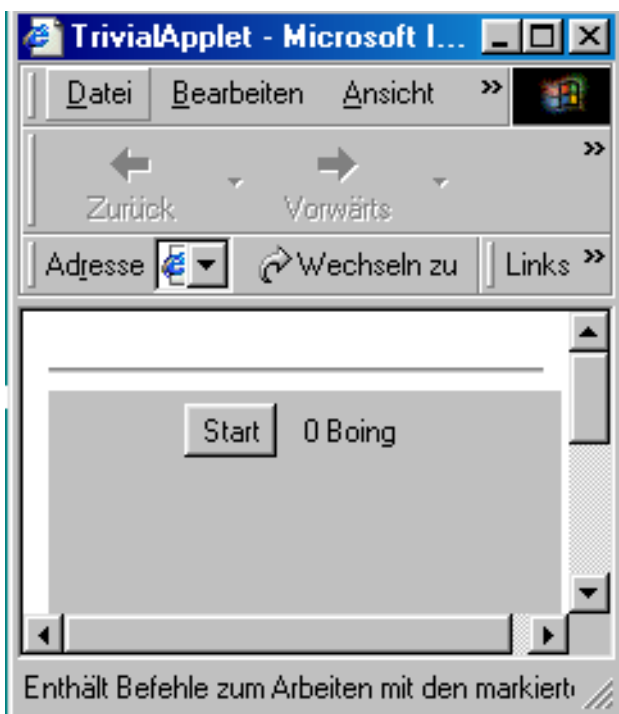
Es ist also doch noch geräte - und systemspezifisch, wie Applets aussehen. Wir nehmen das erst einmal zur Kenntnis und werden im Laufe weiterer Programme sehen, ob man dagegen etwas machen kann.



Applet mit dem Netscape Navigator



Applet im Browser Safari



Applet mit dem Explorer unter WIN 98

In allen Versionen sieht man, dass die beiden Objekte oben in der Mitte des Applet-windows plaziert sind. Weiterhin erkennt man, dass die Schrifteinstellung des Browsers die Größe der Objekte bestimmt und dass die Farbe des Hintergrundes und der Schrift ebenfalls nicht von unserem kleinen Programm bestimmt wurden.

Alle diese Eigenschaften lassen sich im Javaprogramm beeinflussen.

Das ist nicht wirklich wichtig, ist aber einige kleine Versuche wert.

Teste folgende Methoden an den im Appletwindow vorhandenen Objekten aus.

Ein Objekt sichtbar und unsichtbar machen:

```
meinButton1.setVisible(true);
meinButton1.setVisible(false);
```

Man erkennt, dass es hier um eine Methode geht, ein Objekt sichtbar und unsichtbar zu machen. Das geht natürlich mit *meinLabel1* auch.

Man beachte noch einmal die Syntax beim Aufrufen einer Methode:

Objektname.Methode(Parameter);

Genauer wäre es noch hier an Stelle von Objektname zu schreiben: *Name der Instanz der Klasse*.

Farbe ins Spiel bringen:

```
meinButton1.setForeground(Color c);
meinButton1.setBackground(Color c);
meinLabel1.setForeground(Color c);
meinLabel1.setBacvkground(Color c);
```

Für das ganze Appletfenster: `setForeground(Color c);`
`setBackground(Color c);`

Im Beispiel mit dem fliegenden Ball ist die Erzeugung einer Farbe schon erwähnt worden. Hier soll es reichen, die fertig vorgegebenen Werte für die Farbvariable *c* zu benutzen :

white *lightGray* *gray* *darkGray* *black* *red* *blue*
green *yellow* *magenta* *cyan* *orange* *pink*

Diese Farben sind aus der Klassenbibliothek *java.awt.Color* abgeleitet.

Aufgabe: Erzeuge ein Appletwindow mit mindestens 6 beschrifteten Buttons. Färbe auch das ganze Appletwindow. Probiere das Programm - wenn möglich - in verschiedenen Browsern oder Appletviewern aus.

Eine Lösung könnte wie folgt aussehen:

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

```
/**
Hier das Programm VLIN 12, das farbige Buttons und Labels darstellen soll wenn es die virtuelle
Maschine auf Win 98 erlaubt VLIN 12 aus 10/2001 von HGB
*/
```

public class VLIN12 extends Applet

```
{
//*****//
// Zuerst die Festlegung der Instanzen der Klassen Button und Label
//*****//
Button rotButton;
Button blauButton;
Button gelbButton;

// man kann es auch wie folgt schreiben also Aufzählung mit Komma getrennt
Button gruenButton, cyanButton, magentaButton;
```

```
// oder auch gleich mit Initialisierung
Button grauButton = new Button("Grau");
```

```
public void init()
```

```
{
  /**
  // Nun die Objekte initialisieren, wenn das noch nicht geschehen ist
  /**
    rotButton      =new Button("rot");
    blauButton     =new Button("blau");
    gelbButton     =new Button("gelb");
    gruenButton    =new Button("gruen");
    cyanButton     =new Button("cyan");
    magentaButton  =new Button("magenta");
  /**
  // Nun die Objekte mit Vorder - und Hintergrundfarben versehen
  /**
    rotButton.setForeground(Color.red);
    rotButton.setBackground(Color.white);
  /**
  // Auch mal eine Schrift setzen hier Typ ARIAL siehe Anmerkungen
  /**
    rotButton.setFont(new Font("Arial",0,24)); // siehe unten

    blauButton.setForeground(Color.white);
    blauButton.setBackground(Color.blue);
    gelbButton.setForeground(Color.yellow);
    gelbButton.setBackground(Color.black);
    gruenButton.setForeground(Color.black);
    gruenButton.setBackground(Color.green);
    cyanButton.setForeground(Color.black);
    cyanButton.setBackground(Color.cyan);
    magentaButton.setForeground(Color.black);
    magentaButton.setBackground(Color.magenta);
    grauButton.setForeground(Color.black);
    grauButton.setBackground(Color.gray);

  /**
  // Nun die Objekte im Applet unterbringen
  /**
    setBackground(Color.white); //hier für das ganze Appletwindow
    add(rotButton);
    add(blauButton);
    add(gruenButton);
    add(gelbButton);
    add(gruenButton);
    add(cyanButton);
    add(magentaButton);
    add(grauButton);

  }
}
```

Schriften

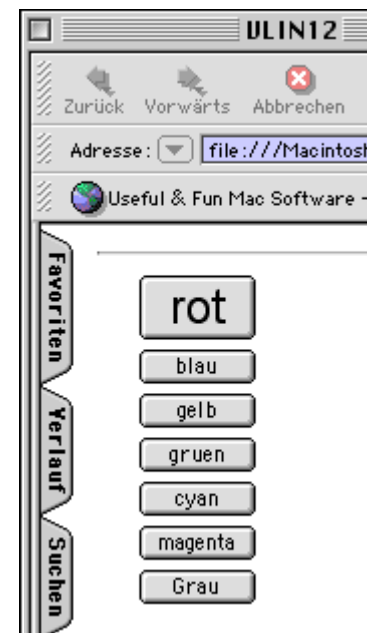
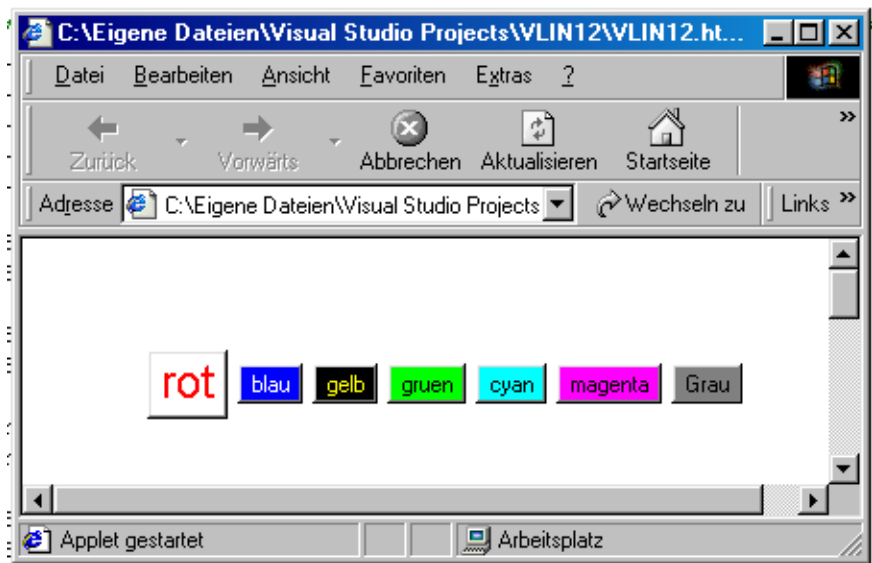
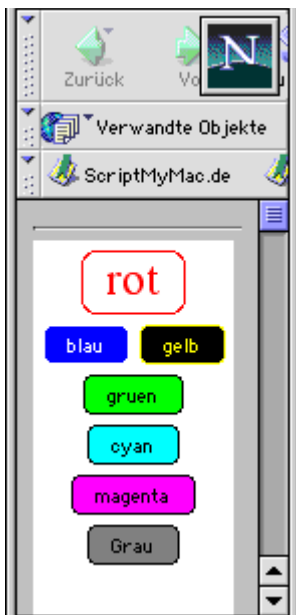
Die Schriftart ist im obigen Beispiel mit `setFont(new Font("Arial",0,24));` gesetzt worden. Dabei ist `Font` eine Klasse aus `java.awt.Font` und `setFont` die Methode, mit der man für eine Komponente des Applets die Schrift setzt.

Dabei gilt für die Parameter, die angegeben werden:

`Font(String name, int style, int size);`

Name der Schriftart Stil mit 0 = standard, 1 = fett und 2= kursiv Größe in Punkt

Wieder sieht das Ergebnis der Bemühungen in verschiedenen Browsern unterschiedlich aus:



Offensichtlich ist wieder allen Lösungen gemeinsam, dass es von der Größe des Appletwindows abhängt, wie die Elemente plaziert werden.

Aufgabe:

Schreibe ein Programm, das 3 Reihen mit je 3 Buttons in einem Fenster zeigt. Wähle dazu die Beschriftungsgröße und die Abmessungen des Appletwindows so, dass diese Anordnung klappt. Bei Bedarf können die Buttons farbig sein.

Wähle als Beschriftung ein "X" oder ein "O".



2.2 Interaktives Ballspielen

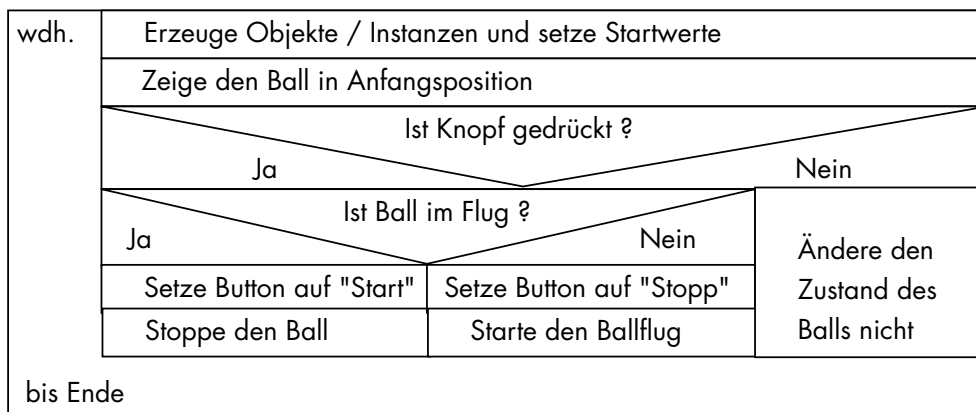
Zurück zum eigentlichen Ziel der derzeitigen Bemühungen.

Das interaktive Programm soll:

- Ein Button mit der Aufschrift "Start" haben und ein Feld, in dem die aktuelle Boinganzahl abzulesen ist.
- Wenn man "Start" klickt, soll der Ball losfliegen.
- Die Beschriftung des Button ändert sich auf "Stopp".
- Bei jeder Reflexion am Rande soll ein Zähler um 1 erhöht werden und dementsprechend wird die Anzeige im Textfeld geändert.
- Wenn man mit der Maus auf den "Stopp"-Button klickt, dann soll der Ball seine Flugbewegungen einstellen.

Das alles muss der Eventmanager für uns erledigen.

Planung des Programms:



Dabei bleiben einige Teile im Moment noch offen. Müssen wir überhaupt etwas tun, wenn der Button nicht gedrückt ist? Läuft dann der Ball nicht von selbst weiter? Was bedeutet eigentlich *wdh. bis Ende*?

Der wichtigste Punkt, der geklärt werden muss, ist der Mausklick auf den Button.

mit `Button meinButton1 = new Button ("Stopp");`

erzeugen wir einen Button, der noch nichts tut.

Um ihn mit Leben zu erfüllen, wird dem Button ein sogenannter *ActionListener* zugeordnet. Das ist ein Objekt, das den Mausklick bearbeitet. Übersetzt also ein Aktionslauscher. Der *ActionListener* hat nur **eine** Methode

```
public void actionPerformed(ActionEvent e)
```

actionPerformed greift dabei auf ein Objekt der Klasse *ActionEvent* zu. In diesem Objekt sind die Einzelheiten des Events (Ereignisses) abzufragen.

Wird nun der Button gedrückt, so sendet er ein *ActionEvent* an seinen Ereignislauscher. Dazu muss dem Knopf vorher ein solcher Lauscher zugeordnet worden sein. Das geschieht mit dem Aufruf.

```
meinButton.addActionListener(StartStopLauscher);
```

Dann bekommt die Methode *actionPerformed* den *ActionEvent* übergeben. Aus dem kann man mit *getActionCommand* die Beschriftung des Button abfragen.

Es ist eine übliche und übersichtliche Arbeitsweise, sich die Klasse, die die Ereignisse behandelt selbst zu basteln und die Klassendefinition nicht innerhalb der Methode *init* des Applets zu positionieren.

Das kann dann etwa folgendermaßen aussehen:

```

/*****/
// Hier wird nun die Klasse konstruiert
//
/*****/
class StartStopLauscher implements ActionListener // implements = eingebaut
{
    public void actionPerformed(ActionEvent meinEvent)
    {
        String befehl; // hält die Knopfbeschriftung fest
        befehl = meinEvent.getActionCommand( ); // hole Knopftitel
        if (befehl.equals("Start")) // equals ist die Methode für
        // Stringvergleiche
        {
            meinButton1.setLabel("Stopp"); // falls der Ball fliegt , zeigt der Button
            flug=true; // jetzt "Stopp" und flug ist true
        }
        else
        {
            meinButton1.setLabel("Start"); // falls der Ball steht ,zeigt der Button
            flug=false; // jetzt "Start" und flug ist false
        }
    }
} // Ende von actionPerformed
// Ende von Start_Stop_lauscher

```

Das sieht doch alles recht kompliziert aus und das ist es auch.

Wichtig !

Man kann auf die eigene Klasse verzichten, und dem ganzen Applet einen *ActionListener* zuordnen. Dann muss die Methode *actionPerformed* dennoch geschrieben werden. Beachten sie die beiden Programmalternativen, die nachfolgend ausführlich kommentiert sind. Auch das Abfragen des Knopfes kann auf andere Weise realisiert werden, als hier geschildert.

Es ist jetzt noch recht unklar, wie diese Klasse im Programm untergebracht werden soll. Es ist auch noch nicht klar, ob und wie das mit dem Ballflug wohl klappen soll. Hier erkennt man aber, dass einen Teil des oben gezeigten Struktogramms das System für uns übernimmt. Die Frage "Ist der Knopf gedrückt ?" müssen wir in unserem Programm nicht mehr behandeln. Das macht der Eventmanager für uns, der überhaupt nur ein passende Reaktion auslöst, wenn der Button gedrückt wurde. Nun werden wir den Programmtext in wohlkommentierten Stücken abarbeiten.

Zuerst werden im Applet alle Variablen und Objekte untergebracht, die benötigt werden. Dabei muss man darauf achten, dass Variable nicht erst in der Methode *init()* vereinbart werden, weil sonst die Klasse *StartStopLauscher* nichts davon weiß.

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

public class Vlin19neu extends Applet
{
    //*****
    // Zuerst alle Variablen und Objekte, die gebraucht werden
    //*****
    Button    meinButton1 = new Button("Start");           // am Anfang auf Start
    Label boing_count = new Label(" 0 Stoesse ");          // am Anfang auf Null
    int    count=0;
    int x,y,vx,vy,r=20,breite,hoehe;                      // Position des Rechtecks, das den Ball einschließt
    boolean flug= false;                                  // fliegt er oder nicht ?
    //*****
    // Nun noch der ActionListener, der unten erst noch als Klasse
    // erzeugt wird. meinLauscher ist dann eine Instanz dieser Klasse
    //*****
    StartStopLauscher meinLauscher= new StartStopLauscher();

    public void init()
    {
        breite = getBounds().width-20;                   // Hole Breite des AppletFensters
        hoehe = getBounds().height-60;                   // Hole Höhe des AppletFensters
        // Starposition und Anfangsgeschwindigkeit des Balls werden per
        // Zufall in vorgegebenen Grenzen erzeugt
        x = (int) Math.round(Math.random()*(breite-50)+25);
        y = (int) Math.round(Math.random()*(hoehe-50)+25);
        vx= (int) Math.round(Math.random()*8-4);
        vy= (int) Math.round(Math.random()*8-4);
        setBackground(Color.white);                      // Hintergrund weiss
        meinButton1.setBackground(Color.yellow);         // Button gelb
        boing_count.setBackground(Color.yellow);         // Label gelb
        meinButton1.addActionListener(meinLauscher);     // der Ereignislauscher für den Button
        add(meinButton1);                                // rein mit dem Button
        add(boing_count);                                // rein mit dem Label
    } // Ende von Init

    public void paint(Graphics g)
    {
        g.setClip(10,60,breite,hoehe);                  // Grafik begrenzen
    }
}
```

```

if (flug== true)                                     // wenn er fliegen soll, dann gehts los
{
    g.setColor(Color.white);
    g.fillOval(x-r,y-r,2*r,2*r);                    // an alter Position löschen
    x = x + vx;                                     // Bewegung in x-Richtung
    y = y + vy;                                     // Bewegung in y-Richtung
    if (x > breite-r-Math.abs(vx) || x <= r+Math.abs(vx))
    {
        vx=-vx;                                     // wenn x-Position ausserhalb, dann umkehren
        count++;                                    // Zähler um eins erhöhen
        boing_count.setText(count+" Stoesse ");      // Label setzen
    }
    if (y > hoehe-r-Math.abs(vy) || y <= r+Math.abs(vy))
    {
        vy=-vy;                                     // wenn y-Position ausserhalb, dann umkehren
        count++;
        boing_count.setText(count+" Stoesse ");
    }
    g.setColor(Color.red);
    g.fillOval(x-r,y-r,2*r,2*r);                    // jetzt malen
    repaint();
}                                                     // Ende von if flug ==true
else
{
    g.setColor(Color.white);                        // Zeichenfläche komplett löschen
    g.fillRect(0,0,breite,hoehe);
    g.setColor(Color.red);                          // Ball malen, wo er gerade ist
    g.fillOval(x-r,y-r,2*r,2*r);
    repaint();
}
} // Ende von paint

//*****
// Hier wird nun die Klasse konstruiert
//*****
class StartStopLauscher implements ActionListener    // implements = eingebaut
{
    public void actionPerformed(ActionEvent meinEvent)
    {
        String befehl;                              // hält die Knopfbeschriftung fest
        befehl = meinEvent.getActionCommand();
        if (befehl.equals("Start"))                 // equals ist die Methode für Stringvergleiche
        {
            meinButton1.setLabel("Stopp");
        }
    }
}

```

```

        flug=true;
    }
else
{
    meinButton1.setLabel("Start");
    flug=false;
}
} // Ende von actionPerformed

} // Ende von start_stop_lauscher
} // Ende von Applet

```



Das Programm funktioniert soweit ganz gut. Es bleiben noch einige kleine Verschönerungen, die wir jetzt angehen.

Das Appletwindow soll noch anders eingeteilt werden. Der Button und das Label sollen unten am Fensterrand erscheinen, der Ball soll nicht durch diese Bedienungselemente hindurch fliegen und oben wäre eine kleine Überschrift nicht schlecht.

Das sind wahrlich keine großen Probleme, die eines Informatikers würdig sind, aber auch so etwas will gemacht sein.



2.3 Der Layoutmanager

In vielen Programmentwicklungssystemen wird die Positionierung von Elementen mit absoluten Koordinaten vorgenommen. Es wäre also denkbar, dass es in Java einen Befehl gibt, der etwa lautet:

```
meinButton.showbutton_(200,120);
```

Der würde dann den Button mit Namen *meinButton* an dem Punkt mit den Koordinaten $x=200$ und $y=120$ zeigen.

Gab es in den ersten Javaversionen aber nicht !

Bevor wir das gemein finden, sei bedacht, dass Javaprogramme auf allen möglichen Computersystemen laufen sollen, die die Verwaltung von Bildschirmkoordinaten sehr unterschiedlich handhaben können. Also muss es anders gehen. In Java erledigt das der Layoutmanager.

Zuerst wird dem Appletwindow ein Layoutmanager zugeordnet, dem werden dann mit dem *add*-Befehl die verschiedenen Objekte zur Plazierung übergeben.

Den Befehl *add* haben wir schon reichlich benutzt. Daraus schließt man mit Recht, dass da schon ein Layoutmanager am Werke war, den wir nicht gesehen haben. Das ist auch tatsächlich der Fall, wie wir gleich sehen werden.

Das einfachste Layout, dass man dem Layoutmanager zuordnen kann ist das

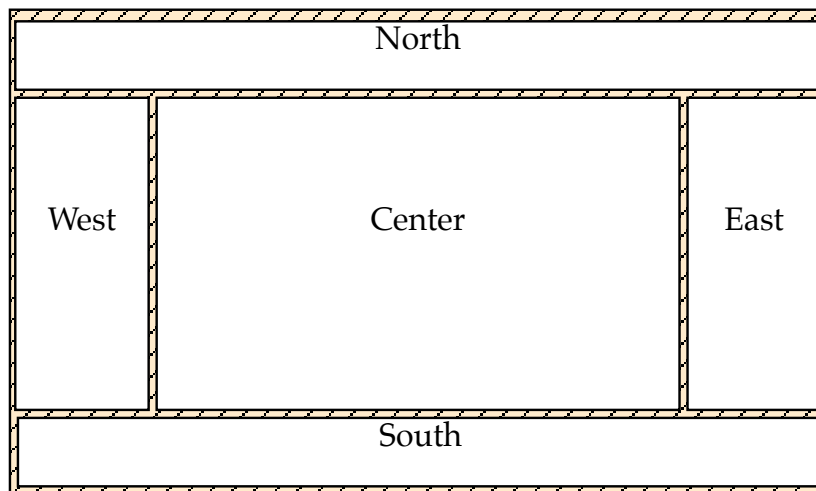
FlowLayout in dem die Elemente in einer Zeile nebeneinander angeordnet werden bis die Zeile voll ist. Danach geht es automatisch in der nächsten Zeile weiter.

Etwas eleganter ist dann das

GridLayout in dem die Objekte in einem Rechteckraster angeordnet werden. Die Anzahl der Zeilen und Spalten muss beim Erstellen des Layoutmanagers angegeben werden.

Eine weitere Möglichkeit ist das

BorderLayout in dem die Objekte auf 5 Bereiche verteilt werden, die nach den Himmelsrichtungen benannt sind (bis auf Center, was aus dem Basketballsport stammt).



CardLayout ist ein Layout, das es möglich macht, mehrere "Unterlayouts" in einem Fenster unterzubringen und nach Bedarf das eine oder das andere zu zeigen.

Es wird klar, dass bei unseren bisherigen Programmen offensichtlich ein FlowLayout existierte. Dazu brauchen wir keine weiteren Übungen. Die Elemente werden in der Reihenfolge ihres Aufrufs mit der *add*-Methode im Fenster positioniert. Das wäre für unsere Absicht nicht geeignet.

Bevor wir aber das Ball-Boing-Programm ändern, sollen durch einige kleine Übungen die Möglichkeiten der anderen Layouts getestet werden.

Ein kleines Programm soll 5 Buttons und 2 Labels im Appletwindow zeigen. Das Programm ist kurz:

```
/*
  ein kleines Demoprogramm mit GridLayout
  VLIN20 aus 10/2001 von HGB
*/
```

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

```
public class VLIN20 extends Applet
```

```

{
    Button meinButton1    =new Button("Knopf 1");
    Button meinButton2    =new Button("Knopf 2");
    Button meinButton3    =new Button("Knopf 3");
    Button meinButton4    =new Button("Knopf 4");
    Button meinButton5    =new Button("Knopf 5");
    Label  meinLabel1     =new Label("Text 1");
    Label  meinLabel2     =new Label("Text 2");

    public void init()
    {
        setLayout(new GridLayout(4,2);    // 4 Zeilen und zwei Spalten
                // keine weiteren Angaben

        add(meinButton1);
        add(meinButton2);
        add(meinButton3);
        add(meinButton4);
        add(meinLabel1);
        add(meinButton5);
        add(meinLabel2);
    }
}

```

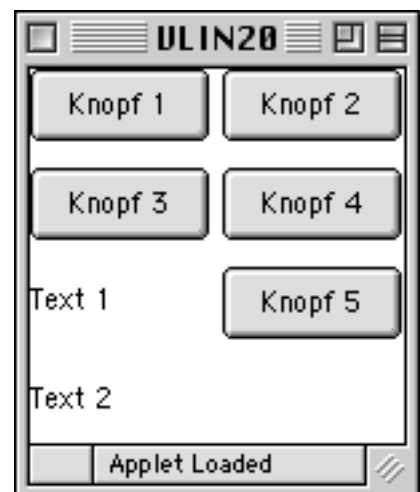
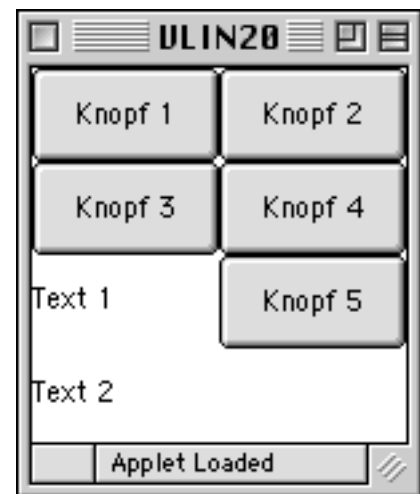
Das Programm zeigt nebenstehendes Ergebnis. Dabei fällt auf, dass man das Fenster in der Größe verändern kann, und sich dann die Größe der Buttons mit verändert. Die Größe der Objekte bestimmt sich also aus der (während der Lebenszeit des Programms veränderbaren) Fenstergröße.

Eine Erweiterung erlaubt es nun noch ,die horizontalen und vertikalen Abstände zwischen den Objekten zu bestimmen.

```
setLayout(new GridLayout(4,2,5,10));
```

erzeugt ein Layout mit 4 Zeilen, 2 Spalten, 5 Punkten Abstand zwischen den Objekten in horizontaler Richtung und 10 Punkten Abstand in vertikaler Richtung.

Nun das Ganze mit dem BorderLayout. Das Programm wird dazu an einigen wenigen Stellen geändert. Es kommt besonders darauf an, anzugeben in welchem Teil ein Objekt plaziert werden soll. Der Ausschnitt des Programms sieht wie folgt aus:



```

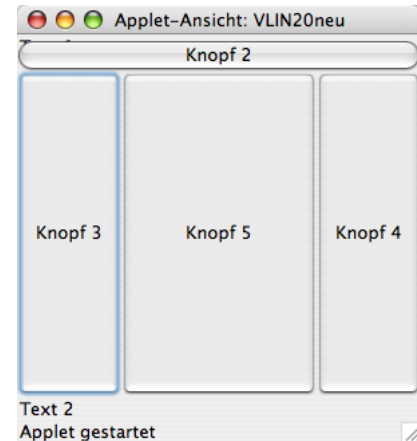
setLayout(new BorderLayout());           // keine weiteren Parameter
add("North",meinButton1);              // mit Angabe der Himmelsrichtung
add("North",meinButton2);
add("West",meinButton3);
add("East",meinButton4);
add("South",meinLabel1);
add("Center",meinButton5);
add("South",meinLabel2);

```

Das Ergebnis zeigt deutlich, dass es wieder auf die Reihenfolge ankommt, in der die *add*-Methode eingesetzt wird. Es wird in einem Teil des Layouts immer nur das zuletzt mit *add* platzierte Objekt angezeigt.

Ausserdem sieht man, dass die West - und Ost-Objekte mit ihrer ganzen Beschriftung dargestellt werden und bei Verkleinerung des Fensters zuerst das Zentrum zu leiden hat. Mit

```
setLayout(new BorderLayout(5,10));
```



kann man wieder einen horizontalen und vertikalen Abstand zwischen den Objekten einführen.

Das NULL-Layout

Hier ist es nun möglich, Buttons und Labels mit der Methode *setBounds* im Fenster zu platzieren:

```
setBounds(linke obere Ecke X, linke obere Ecke Y, Breite, Höhe);
```

Es ist auch möglich, die Objekte mit

```
setLocation(linke obere Ecke X, linke obere Ecke Y);
```

und

```
setSize(Breite,Höhe);
```

im Appletwindow zu plazieren. (Dazu wird es weiter unten noch ein Beispiel geben.) Allerdings bleibt es nun auch dem Programmierer überlassen, darauf zu achten, dass sich die Objekte nicht gegenseitig verdecken (kann man gleich im Beispielprogramm sehen) und das die Objekte auch ins Appletwindow passen.

Das sind nun die Methoden, die das "direkte" Positionieren der Objekte erlauben.

Ein kleines Beispielprogramm

Das oben stehende Programm soll im mittleren Teil verändert werden:

```
setSize (200,250); // Größe des Fensters setzen
setLayout(null); // das null Layout
//*****
// Jetzt kommt Handarbeit. mit der Methode setBounds werden
// die Maße der Buttons angegeben
//*****
meinButton1.setBounds(10,10,80,80); // Koordianten linke obere Ecke, Breite,Höhe
add(meinButton1);
meinButton2.setBounds(95,10,80,80);
add(meinButton2);
meinButton3.setBounds(190,10,80,30);
add(meinButton3);
meinButton4.setBounds(10,200,70,30);
add(meinButton4);
meinButton5.setBounds(95,200,150,30);
add(meinButton5);
meinLabel1.setBounds(10,80,50,15);
meinLabel1.setBackground(Color.red); // nun noch die Labels hier mal rot
add(meinLabel1);
meinLabel2.setBounds(20,97,100,15);
meinLabel2.setBackground(Color.gray); // nun noch die Labels hier mal grau
add(meinLabel2);
```

Verändert man die Größe des Fensters, dann behalten die Objekte dennoch ihre absolute Position bei.

Panels

Ein weiteres Verfahren, Objekte im Appletwindow zu positionieren, sind die *Panels*. Diese Flächen können mit Layouts gefüllt werden und selbst dann in einem weiteren Layout plaziert werden.

Im folgenden Beispiel werden wir noch einige neue Objekte kennenlernen.

Das sind:

TextField ein Textfeld, das eine Zeile Text aufnehmen kann, die editierbar ist !
TextArea ein Textfeld, das mehrere Zeilen Text aufnehmen kann, die editierbar sind.
Canvas eine "Leinwand" , auf der Grafikbefehle ausgeführt werden können.
Choice ein "Klappmenü" mit mehreren Eintragungen aus denen man wählen kann



Doch der Reihe nach:

TextField

```

Deklaration:      TextField      meineZeile;
                  . . . . .
                  meinZeile      = new TextField(22);
    
```

dabei gibt die Zahl in Klammern die Anzahl der Zeichen für die Textzeile an.

Methoden für TextField

meineZeile.**setText**(String *s*); setzt einen Textstring *s* in die Zeile *meineZeile*

String *s* = meineZeile.**getText**(); liefert den Inhalt der Textzeile in den String *s*

meineZeile.**setEditable**(true/false); Textzeile auf editierbar / nicht editierbar setzen

setText, *getText* und *setEditable* sind einige Methoden, mit denen man in Textzeilen arbeiten kann.

TextArea

```

Deklaration:      TextArea      meinFeld;
                  . . . . .
                  meinFeld = new TextArea(12,40);
    
```

```

oder:              meinFeld = new TextArea("Text", 12,40, .TextArea.scroll);
    
```

Hier wird ein mehrzeiliger Textbereich definiert. So eine TextArea kann auf zwei Weisen erzeugt werden (die Klasse hat zwei mögliche Konstruktoren):

Im ersten Fall geben die beiden Zahlen in Klammern an, wieviele Zeilen mit wieviel Zeichen es denn sein sollen. Im obigen Beispiel also 12 Zeilen mit jeweils 40 Zeichen.

Im zweiten Fall kommen noch zwei Angaben hinzu:

Zuerst der Text, der zu Beginn im Feld stehen soll und dann am Ende, welche Art von Scrollbalken am Textfenster erscheinen sollen. Das können folgende vorgegebene Werte sein:

SCROLLBARS_BOTH	horizontale und vertikale Scrollbalken
SCROLLBARS_NONE	keine Scrollbars (auch zu erreichen, wenn man nach Version 1 deklariert)
SCROLLBARS_VERTICAL_ONLY	was könnte das sein ?
SCROLLBARS_HORIZONTAL_ONLY	

Also:

```
meinFeld = new TextArea("Testtext", 12,40,TextArea.SCROLLBARS_BOTH),
```

wäre eine mögliche Festlegung für ein Textfeld mit mehreren Zeilen und zwei Scrollbars .

Methoden für TextArea

meinFeld. append ("Huttitutti");	fügt am Ende des Textfeldes den Text "Huttitutti" an
meinFeld. insert ("Hutti", 12);	fügt an der 12 Buchstabenposition den Text "Hutti" ein

Will man auf den Inhalt eines TextAreas zugreifen, dann muss man einen TextListener installieren, der ähnlich abgefragt werden kann, wie der ActionListener bei einem Button.

Choice

Deklaration:

```
Choice    meineWahl;
.....
meineWahl = new Choice ();
```

Das so erzeugte Auswahlmenü ist noch leer kann dann aber durch folgende Methode mit Inhalten gefüllt werden:

meineWahl. add (" Auswahlpunkt 1");	fügt dem Menü die erste Zeile hinzu
meineWahl. add (" Auswahlpunkt 2");	fügt dem Menü die zweite Zeile hinzu
.....	u.s.w.

Weitere Methoden für Choice

int i= meineWahl.**getSelectedIndex**(); liefert die Nummer des gewählten Menüpunktes und speichert dieses Nummer in einer ganzzahligen Variablen *i*.

String s = meineWahl.**getSelectedItem**(); liefert die Ausgewählte Menüzeile als String

Auch hier ist zu beachten, dass mit der Methode **add** das Auswahlmenü im Appletwindow plaziert werden muss.

Canvas

Eine Malfläche. Die wird gebraucht, wenn in einem Fenster nicht alle Bereiche für die Ausgabe von Grafikbefehlen zur Verfügung stehen sollen

Deklaration:

```
Canvas    meineMalflaeche;
.....
meineMalflaeche = new Canvas ();
```

Nun alles zusammen in einem kurzen Beispiel:

```
//
// VLIN22neu.java
// VLIN22neu
//
// Created by Hans-Georg Beckmann on Thu Feb 19 2004.
import java.awt.*;
import java.applet.Applet;

public class VLIN22neu extends Applet
{
    Button        meinButton1,meinButton2,meinButton3;           // 3 Buttons
    Label         meinLabel1,meinLabel2,meinLabel3;              // 3 Label
    TextField     TF1,TF2;                                       // 2 Textfelder
    TextArea      TA1;                                           // 1 Textarea
    Choice        myChoice;                                       // 1 Auswahlmenü
    Canvas        meineMalflaeche1;                               // ein Malfläche
    Panel         pan1,pan2,pan3;                                 // hier nun Panels

    public void init()
    {
        meinButton1 = new Button("Knopf 1");
        meinButton2 = new Button("Knopf 2");
        meinButton3 = new Button("Knopf 3");

        TF1 =     new TextField("Textfeld 1");
        TF2 =     new TextField("Textfeld 2");
        TA1 =     new TextArea("Eine Textflaeche",6,40,TextArea.SCROLLBARS_VERTICAL_ONLY);

        myChoice = new Choice();
        myChoice.add("Der erste Eintrag");
        myChoice.add("Die zweite Zeile");
        myChoice.add("Die dritte Zeile");

        meinLabel1 = new Label(" Label 1 ");
        meinLabel2 = new Label(" noch ein Label ");
```

```

meinLabel3 = new Label(" Label 3 ");
meineMalflaeche1=new Canvas();
meineMalflaeche1.setSize(220,80);
meineMalflaeche1.setBackground(Color.magenta); // damit man sie auch sieht

```

Nun sind alle Objekte auch mit Leben erfüllt. Man kann versuchen, hier mit der Methode *meineMalflaeche.setSize(Breit,Hoch)*; die Dimensionen des Canvas festzulegen. Man muss aber feststellen, dass es zwar keine Fehlermeldung gibt, aber die Größenzuweisung in einigen Browsern mal wieder nicht klappt¹.

```

// jetzt Panel 1 erzeugen und mit passenden Layouts füllen
// das GridLayout mit 2 x 2 Zeilen x Spalten und dann mit 3 Objekten

    pan1 = new Panel();
    pan1.setLayout(new GridLayout(2,2,5,5)); // eine Stelle bleibt leer
    pan1.add(meinButton1);
    pan1.add(meinButton2);
    pan1.add(meinButton3);
// jetzt Panel 2 erzeugen und mit passenden Layouts füllen
// das FlowLayout mit meinButtonons, Labels und TFs
    pan2 = new Panel();
    pan2.setLayout(new FlowLayout(FlowLayout.LEFT)); // Objekte linksbündig
    pan2.add(TF1);
    pan2.add(meinLabel1);
    pan2.add(meinLabel2);
// jetzt Panel 3 erzeugen und mit passenden Layouts füllen
// das FlowLayout mit meinButtonons, Labels und TFs
    pan3 = new Panel();
    pan3.setLayout(new GridLayout(3,2,5,5));
    pan3.add(meinLabel3);
    pan3.add(myChoice);
    pan3.add(TF1);
    pan3.add(TF2);

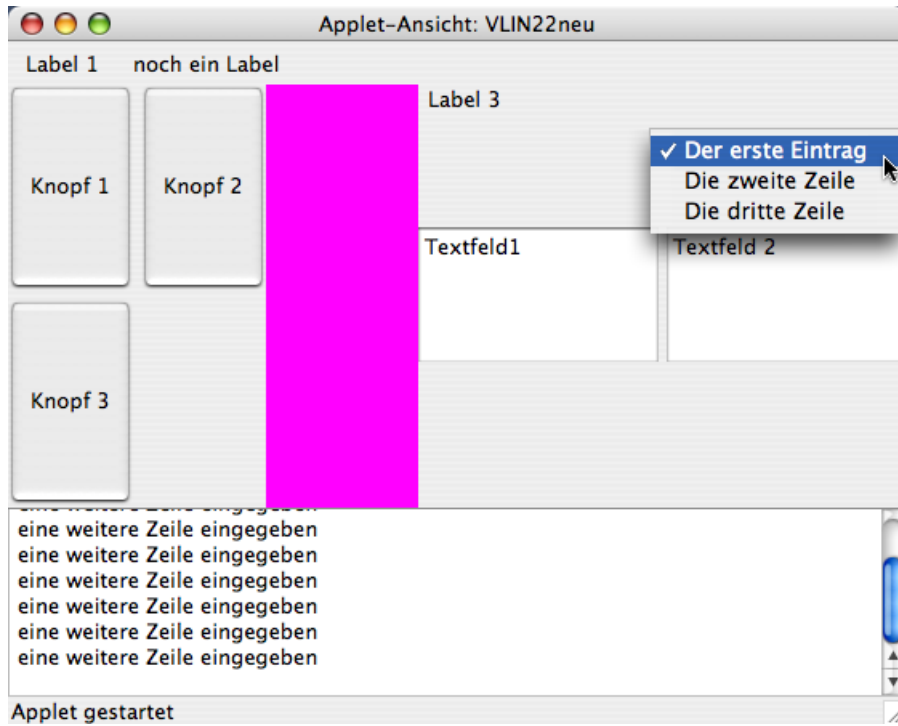
// nun alles in ein BorderLayout zusammenfassen

    setLayout(new BorderLayout());
    add("North",pan2); // Panel 2 in den Norden
    add("West",pan1); // Panel 1 in den Westen
    add("East",pan3); // Panel 3 in den Osten
    add("South",TA1); // TextArea in die Mitte
    add("Center",meineMalflaeche1); // Malfläche in die Mitte
} // Ende von init
} // Ende von Applet

```

¹ Es klappt im Navigator > 6.0 aber nicht in IE 4.5 oder Navigator 4.5

Insgesamt sieht dann das Ergebnis z.B. wie im nachfolgenden Screenshot aus.
Was für ein dämliches Beispiel - so "nützlich und aufregend".
Bedenken sie aber , dass es nur zu Demonstrationszwecken dient.



Aufgabe:

Das schon mehrfach benutzte Ballspiel soll in einem Appletwindow ablaufen, das etwa folgendermaßen aussieht:

Schreibe ein Javaprogramm, das dieses Layout erzeugt.

Ballspiel



Start

0 Stoesse

Probiere einige einfache Grafikbefehle in diesem Applet aus. Was fällt auf ?
Kopiere das Ballprogramm in das Applet und lasse es laufen.
Was muss geändert werden ?

Lösungshilfen

Das Aufteilen des Appletwindows sollte eigentlich keine grossen Probleme bereiten. Dabei ist es nicht sehr wichtig, mit welcher Layoutmethode man die Objekte passend positioniert. Man muss nur immer recht bald probieren, ob denn das Programm im Browser korrekt dargestellt wird.

Wenn alles geklappt hat, dann könnte das Programm bis hierher etwa wie folgt aussehen.

```
//
// VLIN24neu.java
// VLIN24neu
//
// Created by Hans-Georg Beckmann on Thu Feb 19 2004.

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

public class VLIN24neu extends Applet
{
//*****
// Zuerst die Komponenten, die im Appletwindow
// auftauchen sollen
//*****
    Canvas      meineMalflaeche    = new Canvas();      // eine Malfläche
    Button      meinButton1       = new Button();        // ein Button
    Label       meinLabel1        = new Label();         // ein Label
    Label       meinTitel         = new Label();         // ein Label für den Titel
//*****
// in init() werden nun die Objekte mit Attributen
// gefüllt. Wie schon oft gesehen, werden nicht alle
// Farben dann von allen Browsern dargestellt
//
//*****
public void init()
{
//*****
// Zuerst die Malfläche
//
//*****
    meineMalflaeche.setLocation(30, 30,300,210); // linke obere Ecke bei 30,30
    meineMalflaeche.setBackground(Color.cyan); // zu Testzwecken Fläche in CYAN
}
```

```

//*****
// dann der Button
//
//*****
meinButton1.setForeground(Color.white); // Schriftfarbe weiss
meinButton1.setLocation(60, 260,120,20); // linke obere Ecke bei 60, 260
meinButton1.setLabel("Stopp"); // Beschriftung
meinButton1.setBackground(Color.red); // Hintergrund rot
//*****
// dann das Label
//
//*****
meinLabel1.setText("0 Stoesse "); // Anfangsbeschriftung
meinLabel1.setBounds(200, 260,110,20); // linke obere Ecke bei 240,260
meinLabel1.setBackground(Color.yellow); // Hintergrund gelb
meinLabel1.setAlignment(Label.CENTER); // zentriert Beschriften
//*****
// dann das Label für die Überschrift
//
//*****
meinTitel.setText(" Ballflug "); // Anfangsbeschriftung
meinTitel.setLocation(25, 2,290,25); // linke obere Ecke bei 240,260
meinTitel.setBackground(Color.white); // Hintergrund weiss
meinTitel.setFont(new Font("SanSerif",1,18)); // Schriftart und Schriftgröße
meinTitel.setAlignment(Label.CENTER); // zentrierte Ausgabe

//*****/
// und nun das ganze Appletwindow
//
//*****/
setLocation(0, 0); // linke obere Ecke ist 0,0
setLayout(null); // hier das Nulllayout
setSize(408, 299); // Größe 408 x 299
setBackground(Color.white);
add(meineMalflaeche); // nun die Malfläche hinein
add(meinButton1); // .. und der Button
add(meinLabel1); // ..und das Label
add(meinTitel); // ..und die Überschrift
} // Ende von init
*** // hier noch was einfügen
} // Ende vom Applet

```

Fügt man nun noch ein klein wenig "paint" an der Stelle *** dazu ,

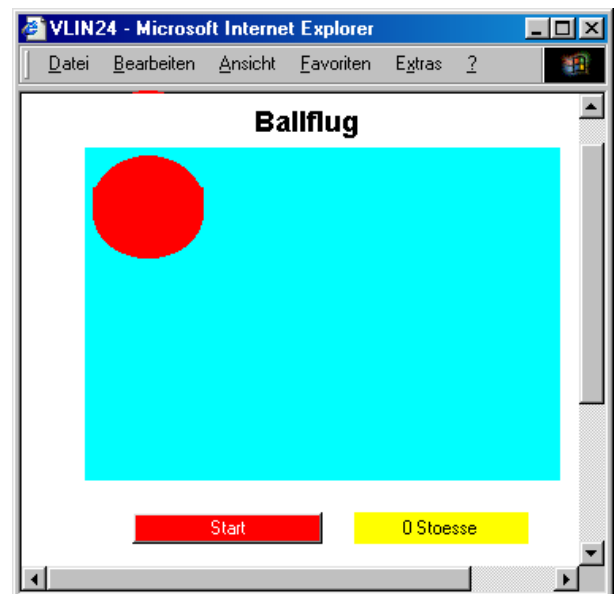
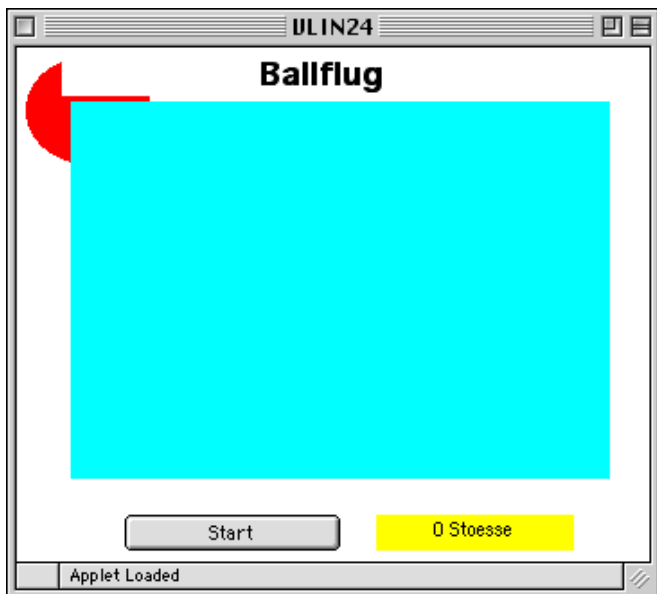
```
public void paint(Graphics g)
{
  ***** // hier auch noch was einfügen

  g.setColor(Color.red);
  g.fillOval(5,5,70,70);
  repaint(); // nicht vergessen,sonst sieht man nichts
}
```

dann muss man feststellen, dass sich die Grafikbefehle nicht auf die Canvasfläche beziehen, sondern immer noch auf das ganze Appletwindow. Der rote Ball ist sogar hinter dem Canvas verschwunden. Man muss also der Methode "paint" noch beibringen, auf welcher Fläche gemalt werden soll.

Fügt man noch die folgende Zeile ein, dann wird offensichtlich dem Grafikobjekt **g** die Malfläche übergeben. Danach muss der Versuch eine Linie zu malen erfolgreich sein.

```
g=meineMalflaeche.getGraphics();
```



ohne den Aufruf: `g= meineMalflaeche.getGraphics();` mit dem Aufruf: `g= meineMalflaeche.getGraphics();`

Nun soll es noch darum gehen, in dieses Programm den Ballflug und die Funktion des Buttons zu integrieren, die wir schon erarbeitet hatten. Zu diesem Zweck ist es natürlich angeraten, die entsprechenden Programmsegmente aus den schon geschriebenen Programmen heraus zu kopieren.

Dabei muss man nur an einigen wenigen Stellen aufpassen:
Bei den Objekten und Variablen kommen hinzu:

```
Canvas      meineMalflaeche = new Canvas();
int         breite, hoehe;
```

Am Anfang von `init` braucht man:

```
meineMalflaeche.setBounds(30,30,300,210);
meineMalflaeche.setBackground(Color.cyan);
meineMalflaeche.setSize(300, 210);
```

Am Ende von `init()` bekommt man dann:

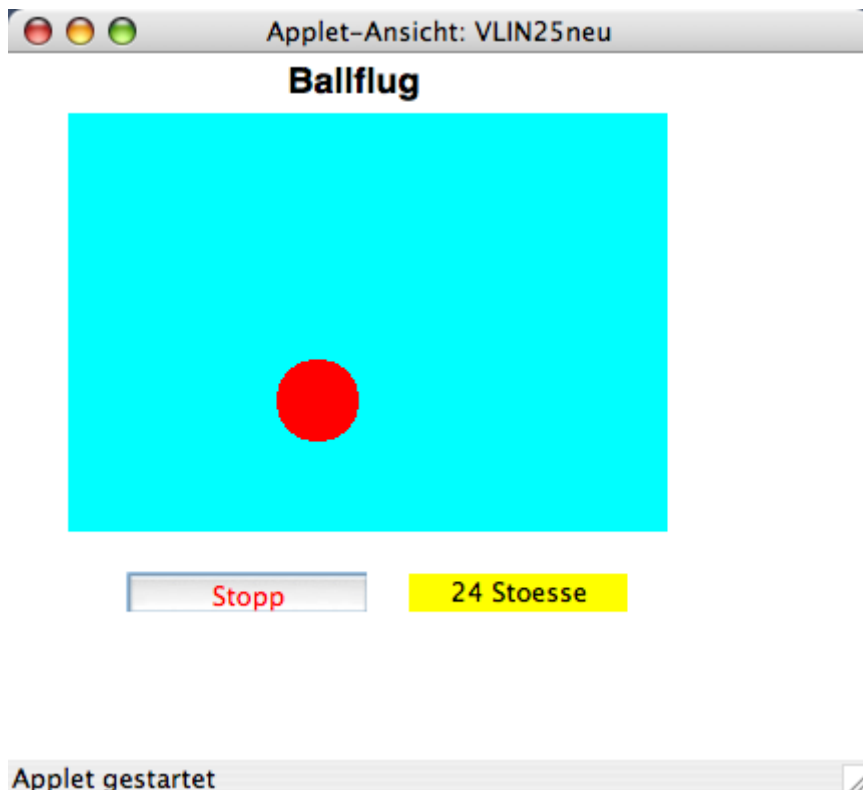
```
breite = meineMalflaeche.getBounds().width;
hoehe = meineMalflaeche.getBounds().height;
```

Mit diesen beiden Zeilen hat man die Breite und die Höhe des Canvas was für die nachfolgende "Flugroutine" gebraucht wird.
Die weiteren Zeilen der Methode `init` sind aus den letzten Programmen bekannt.

Und am Beginn der `paint`-Methode muss man einfügen:

```
public void paint( Graphics g )
{
    g=meineMalflaeche.getGraphics();           // neu

    if (flug== true)                           // wenn er fliegen soll, dann gehts los
    {
        g.setColor(Color.cyan);
    }
}
```



Danach sollte das Ganze zu einem akzeptablen Ergebnis führen.

Nun ist es sicher mit der Interaktivität nicht soweit her bei diesem Beispiel. Außerdem zeigt sich, dass das Programm schon recht komplex ist. Tröstlich ist nur, dass es in Turbo-pascal sehr viel schwieriger gewesen wäre, so etwas zu programmieren.

Aufgaben:

Die Flugfläche soll durch einen Rand begrenzt werden.

Ein zweiter Ball soll auf dem selben Canvas mitfliegen. Er soll eine andere Farbe haben.

Die beiden Bälle sollen beim Zusammenprall einen elastischen Stoß ausführen.

Es soll ein weiterer Button eingebaut werden. Mit dem schon vorhandenen Button soll man den ersten Ball starten/stoppen können mit dem zweiten Button den zweiten Ball starten/stoppen können.

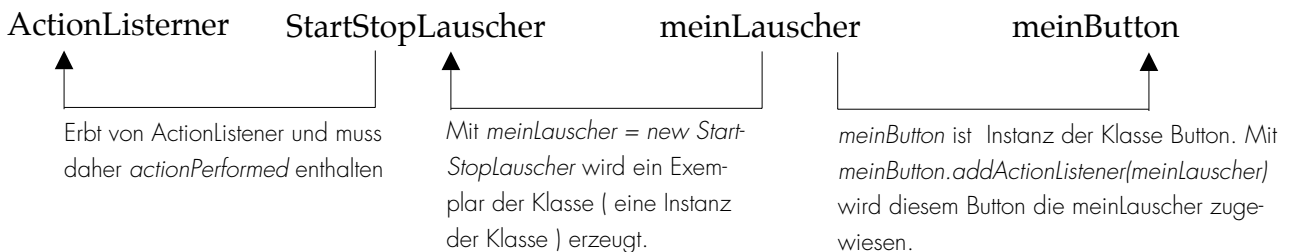
Die Bälle sollen nach jedem Zusammenprall ihre Farbe und ihre Geschwindigkeit zufällig ändern. Dabei darf die Geschwindigkeit nicht Null werden und die Farbe nicht der Hintergrundfarbe entsprechen.

Ergänzungen zum Umgang mit Buttons und ActionListener

In den letzten Beispielen haben wir die umfangreichste und komplizierteste Methode kennengelernt, Buttonevents zu erfassen. Es wurde eine eigene Klasse - StartStopLauscher - erzeugt, die wiederum die Klasse ActionListener implementiert. ActionListener ist eine *abstrakte* Klasse, in der Methodenbezeichnungen vorgegeben sind, ohne dass die Methoden selbst dort schon ein "Innenleben" haben. Solche speziellen Klassen heißen *interface*. Im ActionListener ist es die Methode *actionPerformed*, die dann zwingend von uns geschrieben werden muss.

Im Schema:

Das *interface*



Vereinfachung 1:

In der Methode *actionPerformed* kann man den Button, der mit der Maus geklickt wurde, auch einfacher abfragen, als oben geschildert.

```

public void actionPerformed ( ActionEvent e)
{
    Object welcherKnopf;           // eine Instanz der Klasse Object, die auch Buttons umfasst
    welcherKnopf = e.getSource(); // in der Klasse ActionEvent gibt es die Methode getSource(),
                                   // die die Quelle eines Events als Object angibt
    if ( welcherKnopf == meinButton) // nun kann man direkt fragen, ob es unser Button war, der gerufen hat
    {
        // hier Anweisungen
    }
} // Ende von actionPerformed
  
```

Wir können also auf die Abfrage der Buttonbeschriftung leicht verzichten. Geeignet ist diese Version für das Abfragen mehrerer Knöpfe, für die dann je eine If-Abfrage formuliert wird.

Vereinfachung 2:

Da wir in unserem Beispiel nur einen Knopf haben, der einen Event erzeugen kann, können wir uns auch die Abfrage des Objects sparen und direkt schreiben:

```

public void actionPerformed ( ActionEvent e)
{
    // hier Anweisungen hinein
} // Ende von actionPerformed
  
```

Vereinfachung 3:

Wenn man sich nicht die Mühe machen will, eine eigene Klasse zu erzeugen, die von ActionListener erbt, dann kann man gleich das ganze Applet von ActionListener erben lassen:

```
public class VLIN25neu extends Applet implements ActionListener
{
    .....
```

Die Kopfzeile des Applets wird also nur etwas erweitert und schon hat das ganze Applet seinen ActionListener. Damit muss natürlich auch die Methode *actionPerformed* im Applet auftreten.

Man schreibt sie - wie oben angegeben - z.B. hinter die Methode *paint*.

Man kann nun auf *meinLauscher* und die Klasse *StartStopLauscher* verzichten. Bleibt nur noch, dafür zu sorgen, dass der Button einen ActionListener zugeordnet bekommt.

Das geht etwa so: "nimm den ActionListener, der diesem Applet zugewiesen wurde" --->

```
meinButton.addActionListener(this);
```

Die Zuweisung erfolgt also über das Schlüsselwort **this**.

Aufgabe:

Probieren sie die verschiedenen Versionen im Ballflugprogramm aus.

Eine (erste) eigene Klasse

Im Anschluss an den Ballflug passt es ganz gut, eine Klasse zu erzeugen, die nicht irgend eine fertige Standardklasse beerbt, sondern ganz und gar "neu" ist. Außerdem behandeln wir gleich noch zwei weitere Interfaces mit, die sich mit den Mausevents befassen. Genauer zu diesen Mausereignissen finden sie noch im **Abschnitt 1.5**, wenn mit der Maus gemalt wird.

Wir werden hier nicht die speziellen Klassen (Adapterklassen) verwenden, die im genannten Abschnitt vom JBuilder vorgegeben sind, sondern uns auf die Standardinterfaces beziehen, die natürlich auch unter JBuilder laufen.

Unser Objekt soll eine Kreisscheibe sein, die eine nette Farbe haben kann und einen freundlichen Radius, die man irgendwo am Bildschirm plazieren kann und die man mit der Maus bewegen können soll . Das ist dann auch wirklich mal ein Objekt, das man richtig "anfassen" kann. Es ist daher auch für den Unterricht ein gutes Beispiel für eine Klasse.

Für die übersichtliche Darstellung von Objekten / Klassen verwendet man Klassendiagramme auf die im nächsten Kurshalbjahr noch ausführlich eingegangen wird.

Wir schreiben Attribute (veränderbare oder auch feste Werte / Variablen) und Methoden, die zu einer Klasse gehören sollen übersichtlich auf und vermerken jeweils, ob auf diese von außen zugegriffen werden darf (dann ist das *public*) und welche Parameter ggf. an die Methoden zu übergeben sind.

Das folgende Schema erklärt sich fast von selbst, wenn man weiß, das für *public* ein + - Zeichen verwendet wird.

Klassendiagramm für Scheibe:

Klasse Scheibe
<i><< Attribute >></i>
+ xm,ym : int
+ farbe : Color
+ radius: int
- g: Graphics
<i><< Konstruktor >></i>
+ Scheibe (int x, int y , int r)
<i><< Methoden >></i>
+ setColor(Color c)
+ male()
+ loesche()
+ getroffen (int x, int y):boolean
+ move (int xneu, int yneu)

Name der Klasse
xm und ym = Koordinaten des Mittelpunktes, ganze Zahlen
eine Variable vom Typ Color (genauer: eine Instanz der Klasse ...)
der Radius
brauchen wir zum Malen, muss nicht public sein
mit new Scheibe (20,40, 20) wird eine Scheibe mit r= 20 an der Stelle (20,40) erzeugt. Man sieht sie aber noch nicht !
Farbe setzen
malen an der Stelle (x,y)
loeschen an der Stelle (x,y)
klickt die Maus bei (x,y) und dort ist die Scheibe, dann true
bewege die Scheibe zu einem neuen Punkt

Dieses Klassendiagramm lässt sich fast direkt in Java umschreiben:

```

//*****
// Nun die Klasse Scheibe
//*****
public class Scheibe
{
    int xm,ym;           // Attribute sind public, wenn nichts anderes angegeben ist
    Color farbe= Color.white; // Die Mitte der Scheibe
    int radius;         // Die Farbe der Scheibe
    Graphics g;        // der Radius der Scheibe
    boolean sichtbar;  // für die interne Zeichenmethode
                       // ist die Scheibe gerade zu sehen oder nicht ?
//*****
// Der Konstruktor hat den selben Namen, wie die Klasse
//*****
public Scheibe(int x, int y, int r)
{
    xm=x;           // Von außen angegebene Werte sind x,y und r
    ym=y;           // die werden hier nach innen weitergegeben
    radius=r;
} // Ende vom Konstruktor

```

```

//*****
// Die Methoden
//*****
public void setColor(Color c)
{
    farbe=c;
} // Ende von setColor

public void male()
{
    g=getGraphics();
    g.setColor(farbe);
    g.fillOval(xm-radius/2,ym-radius/2,radius,radius);
} // Ende von male

public void loesche()
{
    g=getGraphics();
    g.setColor(Color.white);
    g.fillOval(xm-radius/2,ym-radius/2,radius,radius);
} // ende von loesche

public boolean getroffen(int x,int y)
{
    if((Math.abs(x-xm)<radius/2)&&(Math.abs(y-ym)<radius/2))
        return true;
    else
        return false;
} // Ende von getroffen

public void move(int xneu, int yneu)
{
    loesche();
    xm=xneu;
    ym=yneu;
    male();
} // Ende von bewegen der Scheibe

} // Ende von Scheibe

```

Studieren sie die Klasse ein wenig. In der Methode *gettroffen* wird der Abstand des übergebenen Punktes von der Mitte der Scheibe errechnet. Ist er kleiner als der Radius, gilt die Scheibe als getroffen.

Nun kann es an den Rest des Applets gehen.

```
//
// Created by Hans-Georg Beckmann on Sun Sep 15 2002. und 22.2.2004
//
```

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

```
public class Scheibenschieber extends Applet implements MouseListener, MouseMotionListener
{
```

Es werden zwei Interfaces implementiert. Der *MouseListener* kümmert sich um die Position der Maus und den Zustand der Maustaste. Der *MouseMotionListener* ist für die Mausbewegungen zuständig. Beide Interfaces haben eine Reihe von "Pflicht"methoden, die in jedem Falle im Applet vorkommen müssen, auch wenn sie möglicherweise leer bleiben.

```
    Color ff=new Color(255,0,0);           // Farbe
    Color ff2=new Color(0,255,255);       // noch eine Farbe
    Scheibe plopp=new Scheibe(50,50,40);  // die erste Scheibe
    Scheibe plopp2=new Scheibe(200,200,60); // die zweite Scheibe
    int x,y;                               // für die Koordinaten der Maus

    public void init()
    {
        setLayout (null);
        addMouseListener(this);           // das Applet bekommt die beiden Interfaces
        addMouseMotionListener(this);     // zugeordnet
        plopp.setColor(ff);               // Farben setzen
        plopp2.setColor(ff2);
    }
```

Die Methode `paint` könnte eigentlich komplett entfallen. Sie tut hier nichts anderes, als die Appletfläche mit Weiß zu füllen. Die Standardfarbe im `JBuilder` ist leider grau, aber die Methode `loesche()` in der Klasse `Scheibe` malt weiß. Das passt also nicht gut zusammen, daher hier die Methode `paint`. Sie können aber auch in `loesche()` grau malen lassen und hier auf `paint` verzichten. Machen sie doch, was sie wollen !

```
public void paint (Graphics g)
{
    g.setColor(Color.white);
    g.fillRect(0,0,400,400);
}
```

Nun die Methoden, die auftauchen müssen:

```
//*****
// Zuerst die Methoden vom MouseListener
//*****
public void mousePressed(MouseEvent e)
{
    x=e.getX();
    y=e.getY();
    plopp.male();
    plopp2.male();
}
```

Wenn die Maustaste gedrückt wird, besorgen wir uns mit e.getX() und e.getY() die Koordinaten, des Punktes, an dem die Taste gedrückt wurde.

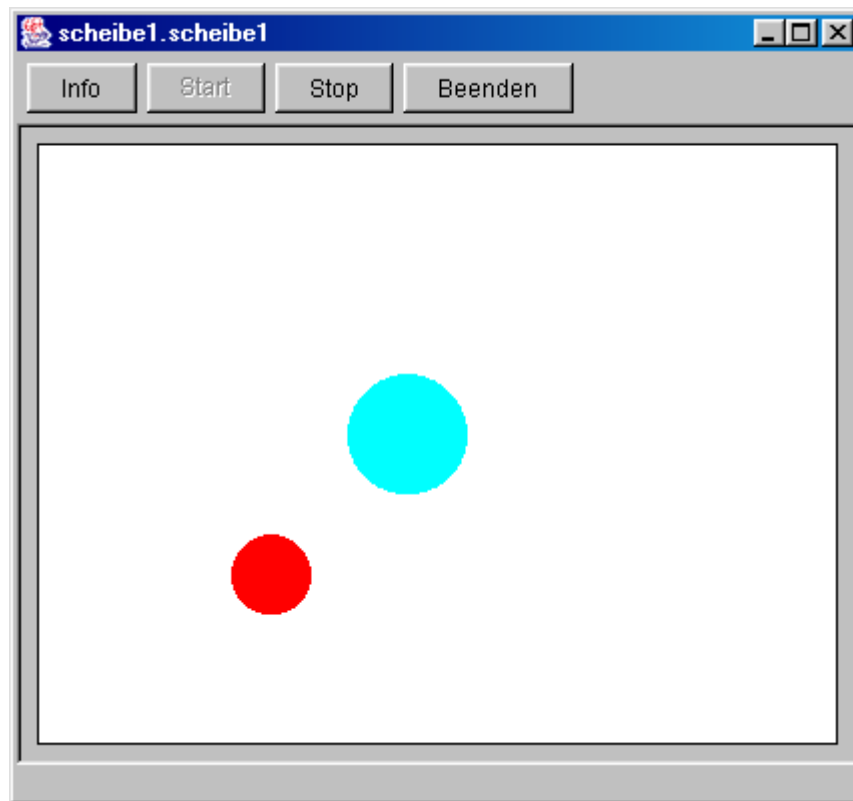
```
public void mouseReleased(MouseEvent e) // Das Loslassen ist hier nicht wichtig
{}
public void mouseEntered(MouseEvent e) // Maus betritt ein Objekt / Bereich - nicht wichtig
{}
public void mouseExited(MouseEvent e) // Maus wieder raus
{}
public void mouseClicked(MouseEvent e) // Maustaste drücken und gleich wieder loslassen
{}
//*****
// Dann die Methoden vom MouseMotionListener
//*****
public void mouseMoved(MouseEvent e) // Maus bewegt sich ohne gedrückte Taste
{}
}
```

Das mausbewegen mit gedrückter Taste ist nun interessant, weil wir damit unsere Scheiben schieben wollen.

```
public void mouseDragged(MouseEvent e)
{
    if(plopp.getroffen(e.getX(),e.getY())) // wenn Scheibe plopp getroffen wurde...
        plopp.move(e.getX(),e.getY()); // bewege sie an die Stelle, die man aktuell mit getX und
                                        // getY abfragt
    if(plopp2.getroffen(e.getX(),e.getY())) // genauso für Scheibe plopp2
        plopp2.move(e.getX(),e.getY());
}
```

Der Eventmanager ruft diese Methode andauernd auf, wenn die Maustaste noch gedrückt ist, und die Maus ihre Position ändert, wodurch sich auch jedesmal für e.getX() und e.getY() neue Werte ergeben.

Nach diesen Methoden wird noch die oben angegebene Klassendefinition eingefügt und natürlich die abschließende Klammer des Applets nicht vergessen.
So simpel das alles aussieht - es funktioniert. Sogar der JBuilder macht hier mal keine Probleme.



Aufgabe:

Schreiben sie auf Basis dieses Applets das Ballspielprogramm noch einmal. Lassen sie diesmal ein Objekt - eine Instanz - der Klasse Scheibe fliegen. Fügen sie der Klasse eine Methode hinzu, die feststellt, ob sich der Ball am Rande des Bildschirms befindet und dann ggf. für die Reflexion sorgt.