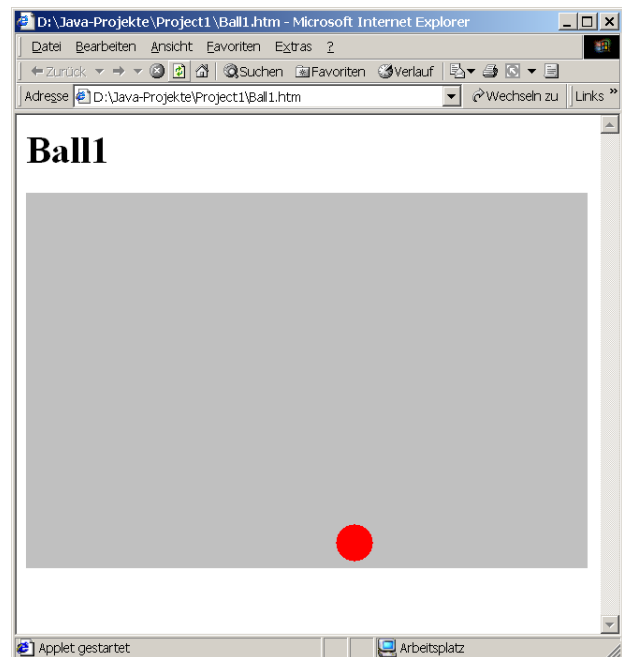
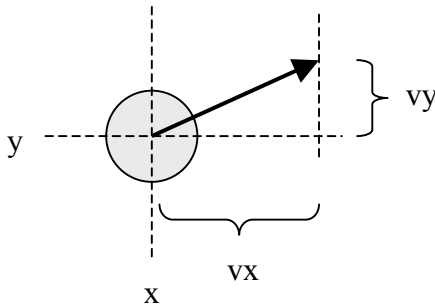


## 1. Bälle

Wir wollen einen Ball auf dem Bildschirm „herumfliegen“ lassen. Ein Ball „fliegt“, wenn er im Laufe der Zeit seine Position auf dem Bildschirm ändert. Wir müssen deshalb diese Position als Variable speichern, z.B. in den Variablen  $x$  und  $y$ . Die Änderung dieser Position pro Zeitintervall können wir als Geschwindigkeit des Balls interpretieren. Zum Ball gehören damit auch die Geschwindigkeitskomponenten  $v_x$  und  $v_y$ .



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

```
public class Ball1 extends Applet
{
    int x,y,vx,vy,r=20,breite,hoehe,;
```

```
public void init()
{
    breite = getBounds().width;
    hoehe = getBounds().height;
    x = (int) Math.round(Math.random()*(breite-50)+25);
    y = (int) Math.round(Math.random()*(hoehe-50)+25);
    vx= (int) Math.round(Math.random()*20-10);
    vy= (int) Math.round(Math.random()*20-10);
}
```

```
public void paint(Graphics g)
{
    x = x + vx;
    y = y + vy;
    g.setColor(Color.red);
    g.fillOval(x-r,y-r,2*r,2*r);
    repaint(25);
}
```

Variable zur Beschreibung  
des Balls und der „Wände“

Appletmaße  
feststellen

Anfangswerte innerhalb  
der Applet-Grenzen  
zufällig wählen

Position ändern

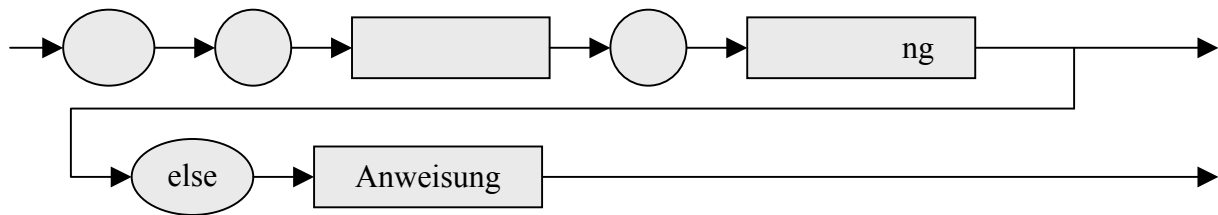
roten Ball zeichnen

neu zeichnen

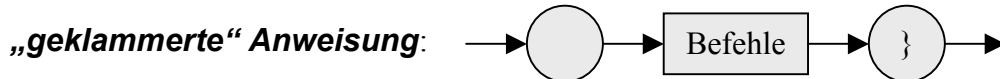
Der Ball „fliegt“ prima, allerdings mit ein paar kleinen Schönheitsfehlern: Er verlässt das Fenster am Rand ohne umzukehren und das Bild flimmert stark.

Wenden wir uns zuerst der „Reflexion“ des Balls an den Wänden zu: Ein Ball wird reflektiert, indem die entsprechende Geschwindigkeitskomponente ihr Vorzeichen ändert. Aus  $v_x$  wird z.B.  $-v_x$  und daraus später  $-(-v_x)=v_x$  usw. Der Ball ist an der Wand, wenn er sich „nahe genug“ an den Begrenzungen unseres Applets befindet. Das können wir mit einer entsprechenden Alternative überprüfen: `if (... Ball an der Wand ...) { ... tue was... }`

Allgemein muss eine **Alternative** der folgenden Syntax gehorchen:



Eine Anweisung kann wie immer aus mehreren Befehlen bestehen, die dann mit geschweiften Klammern „zusammengebunden“ werden:



Zu beachten ist, dass eine Alternative sowohl einseitig (nur die Anweisung nach *if* wird ausgeführt, wenn die Bedingung erfüllt ist) wie auch zweiseitig (es existiert eine echte Alternative nach *else*, die ausgeführt wird, wenn die Bedingung nicht erfüllt ist) gestaltet werden kann.

Für unsere rechte Wand lautet die Bedingung fürs Umkehren:

```
if ( x > breite-r ) vx=-vx;
```

Für die linke Wand könnte man eine ähnliche Bedingung formulieren, die aber leider zu Fehlern führt, wenn wir nicht berücksichtigen, dass die Geschwindigkeit dort negativ ist. Wir müssen deshalb deren Absolutwert verarbeiten, den uns wieder das *Math*-Objekt liefert. Weil es zwei Bedingungen für die Geschwindigkeitsumkehr gibt, verknüpfen wir beide mit dem **logischen ODER** (`||`).

```
if ( x > breite-r-Math.abs(vx) || x <= r+Math.abs(vx) ) vx=-vx;
if ( y > hoehe-r-Math.abs(vy) || y <= r+Math.abs(vy) ) vy=-vy;
```

Die *paint()*-Methode für den umkehrenden Ball lautet dann:

```
public void paint(Graphics g)
{
    x = x + vx;
    y = y + vy;
    if ( x > breite-r-Math.abs(vx) || x <= r+Math.abs(vx) ) vx=-vx;
    if ( y > hoehe-r-Math.abs(vy) || y <= r+Math.abs(vy) ) vy=-vy;
    g.setColor(Color.red);
    g.fillOval(x-r,y-r,2*r,2*r);
    repaint(25);
}
```

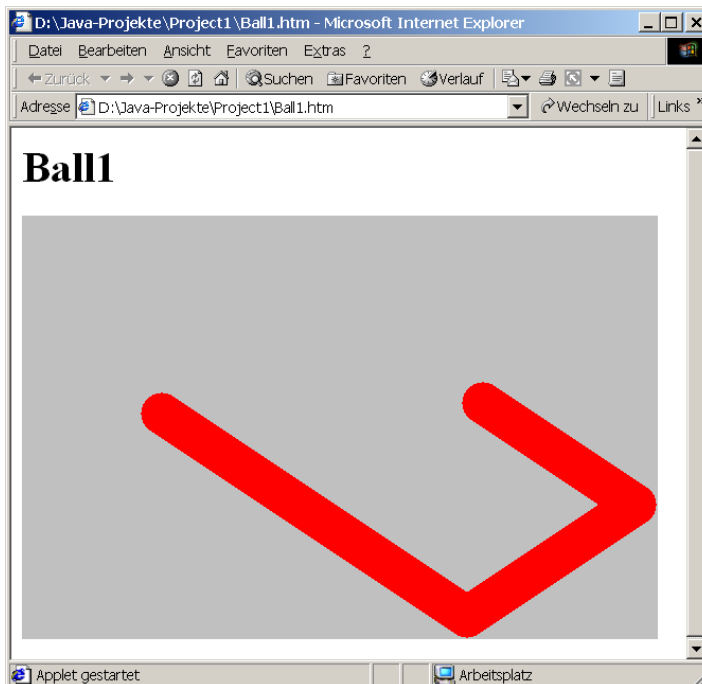
Jetzt müssen wir noch das Flimmern reparieren – und das ist etwas komplizierter. Java löscht normalerweise jedes Mal den Zeichenbereich des Applets, wenn die *Paint()*-Methode aufgerufen wird. Der Befehl dafür ist aber gar nicht zu finden. Das liegt daran, dass immer vor dem *paint()*-Aufruf automatisch eine andere Zeichenmethode namens *update()* aufgerufen wird, die den Zeichenbereich löscht. Wenn wir das nicht wünschen, dann **überschreiben** wir die alte *update()*-Methode durch eine eigene, die nur *paint()* aufruft.

```
public void update(Graphics g)
{
    paint(g);
}
```

Die Aufrufkette zur Grafikauffrischung lautet also: *repaint()* → *update()* → *paint()*

Das Ergebnis ist entsprechend: Der Ball wird nicht mehr automatisch gelöscht, und weil wir nicht selbst dafür gesorgt haben, hinterlässt er eine breite Spur. Der können wir aber wenigstens ansehen, dass es inzwischen mit der Reflexion gut klappt.

Um den Ball zu löschen, benutzen wir den schon bekannten *Exklusiv-Oder-Zeichenmodus*, der (etwas vereinfacht gesagt) die Komplementärfarbe des Untergrunds zum Zeichnen benutzt. Der Trick dabei ist, dass zweimaliges Zeichnen den alten Zustand wieder herstellt. In einer booleschen Variablen *sichtbar* merken wir uns, ob der Ball gerade auf dem Bildschirm zu sehen ist. (Das unten benutzte Ausrufungszeichen vor einem Wahrheitswert verneint den Wert.) Die Position des Balls wird schlauerweise nur dann verändert, wenn er gerade nicht sichtbar ist. (Warum wohl?) Wenn der Ball frisch gezeichnet ist, dann machen wir eine längere Pause vor dem Neuzeichnen, als wenn er gelöscht wurde. Damit flimmert das Bild etwas weniger.



```
public void paint(Graphics g)
{
    g.setColor(Color.red);
    g.setXORMode(Color.white);
    if (!sichtbar)
    {
        x = x + vx;
        y = y + vy;
        if (x > breite-r-Math.abs(vx) || x <= r+Math.abs(vx)) vx=-vx;
        if (y > hoehe-r-Math.abs(vy) || y <= r+Math.abs(vy)) vy=-vy;
        g.fillOval(x-r,y-r,2*r,2*r);
        repaint(50);
        sichtbar=true;
    }
    else
    {
        g.fillOval(x-r,y-r,2*r,2*r);
        repaint();
        sichtbar=false;
    }
}
```

!sichtbar  
entspricht „nicht“ sichtbar

Das Bild flimmert aber immer noch!

Wie führen deshalb alle Zeichenvorgänge in einem „versteckten“ Bild im Hintergrund durch, und erst wenn wir damit fertig sind, stellen wir dieses Bild auf dem „richtigen“ Bildschirm dar. Das Verfahren nennt sich **Doppelpufferung**.

```

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

public class Ball1 extends Applet
{
    int x,y,vx,vy,r=20,breite,hoehe;
    boolean sichtbar;
    Image im;
    Graphics bg;

    public void init()
    {
        breite = getBounds().width;
        hoehe = getBounds().height;
        x = (int) Math.round(Math.random()*(breite-50)+25);
        y = (int) Math.round(Math.random()*(hoehe-50)+25);
        vx= (int) Math.round(Math.random()*20-10);
        vy= (int) Math.round(Math.random()*20-10);
        sichtbar=false;
        im = createImage(breite,hoehe);
        bg = im.getGraphics();
        bg.setColor(Color.white);
        bg.fillRect(0,0,breite,hoehe);
    }

    public void paint(Graphics g)
    {
        bg.setXORMode(Color.white);
        if (!sichtbar)
        {
            bg.setColor(Color.red);
            bg.fillOval(x-r,y-r,2*r,2*r);
            x = x + vx;
            y = y + vy;
            if (x > breite-r-Math.abs(vx) || x <= r+Math.abs(vx)) vx=-vx;
            if (y > hoehe-r-Math.abs(vy) || y <= r+Math.abs(vy)) vy=-vy;
            bg.fillOval(x-r,y-r,2*r,2*r);
            sichtbar=true;
            repaint(0);
        }
        else
        {
            bg.setColor(Color.red);
            bg.fillOval(x-r,y-r,2*r,2*r);
            g.drawImage(im,0,0,this);
            sichtbar=false;
            repaint(10);
        }
    }

    public void update(Graphics g)
    {
        paint(g);
    }
}

```

verstecktes“ Bild *im* vom Typ Image und der dazugehörige Grafikkontext *bg*

s Bild der richtigen Größe n und Grafikkontext setzen

alle Zeichenvorgänge erfolgen in *bg*

alle Zeichenvorgänge erfolgen in *bg*

verstecktes Bild auf dem hirm darstellen

verstecktes Bild auf dem Bildschirm darstellen

## 2. Aufgaben

1. Ändern Sie das Ball-Programm so, dass der Ball eine „Spur“ auf dem Bildschirm hinterlässt.
2. Führen Sie eine „Schwerkraft“ ein, die den Ball nach unten „plumpsen“ lässt. Lassen Sie automatisch in einem Teil des Bildes Weg-Zeit- bzw. Geschwindigkeits-Zeit-Diagramme der Ballbewegung erstellen.
3. Führen Sie eine „Reibungskraft“ ein, die den Ball bremst. Lassen Sie wieder Diagramme erstellen.
4. Nur für Physiker: Führen Sie eine „Lorentzkraft“ ein, die „geladene“ Bälle ablenkt. Simulieren Sie so eine „Blasenkammer“ mit geladenen Teilchen unterschiedlicher Masse.
5. Führen Sie einen zweiten Ball ein. Wenn sich die Bälle treffen, dann sollen sie von einander abprallen.
6. Führen Sie einen dritten Ball ein. Auch hier soll die Simulation halbwegs realistisch sein.
7. Überlegen Sie sich Möglichkeiten, „viele“ Bälle zu simulieren, ohne beim Schreiben einen Krampf zu bekommen. Welche Probleme im Detail treten auf?
8. Führen Sie einen „Billardstock“ ein, mit dem Sie Bälle „anstoßen“ können. Realisieren Sie Billardspiele unterschiedlicher Art (Pool-Billard, ...).
9. Führen Sie einen Korb ein und eine Möglichkeit, den Ball mit unterschiedlicher Geschwindigkeit in unterschiedlicher Richtung abzuwerfen. Spielen Sie damit Basketball.
10. Machen Sie Stoßversuche mit Bällen gleicher Masse. Führen Sie die Stöße zentral bzw. versetzt ausführen.
11. Führen Sie Bälle unterschiedlicher Masse ein. Versuchen Sie, die Stöße zwischen den Bällen realistisch zu simulieren.
12. Benutzen Sie mehrere kleine Bälle und einen großen. Simulieren Sie damit die Brownsche Bewegung
14. Machen Sie Fallversuche mit automatischer Diagrammerstellung. Scannen Sie dazu ein entsprechendes Bild aus dem Physikbuch. Löschen Sie abgebildeten Ball und lassen Sie diesen danach programmgesteuert geeignet „fallen“.

