# 3 Computergrafik

Im folgenden Kapitel wollen wir einige wesentliche Sprachelemente von Java etwas systematischer als bisher einführen. Wir wählen dafür wieder Grafikprogramme, weil sich damit leicht recht hübsche Effekte erzielen lassen und – vor allem –, weil viele Fehler, die man fast immer bei der Programmentwicklung macht, direkt am Bildschirm sichtbar werden. Eingeführt werden

- *Java-Namen* und deren *Gültigkeitsbereiche*.
- *Konstante* und *Variable*
- *Fallunterscheidungen* und *Schleifen*

## 3.1 Ein Kontext zum Zeichnen – das Graphics-Objekt

Einige Java-Komponenten bieten die Möglichkeit, grafische Operationen auszuführen. Verwirklicht wird dieses durch *Graphics*-Objekte, die eine Umgebung (einen *Kontext*) mit entsprechenden Werkzeugen bereitstellen. Zwei davon – die *setColor*- und die *drawLine*-Methode – kennen wir schon. Zu diesen Komponenten gehören die *Frames* (die späteren Bildschirm-Fenster), *Panels* und auch *Applets*.

Zum Zeichnen benutzt man geeignete Werkzeuge, und diese werden natürlich auch von Java in Form von Methoden bereitgestellt. Der *drawLine*-Befehl z. B. zeichnet eine Line.

*drawLine(int,int,int,int)*

Er erwartet in den Klammern vier ganze Zahlen (Typ *int*) als Argumente, die die Koordinaten der Endpunkte der zu zeichnenden Linie angeben. Woher weiß man aber, welche Reihenfolge einzuhalten ist? Dafür gibt es die *Javadokumentation*, in der die Methoden beschrieben sind. Viele Entwicklungsumgebungen zeigen diese beim Drücken der Taste *F1* direkt an, wenn sich der Cursor im gesuchten Begriff befindet.

In *Eclipse* erhält man eine Kurzhilfe beim Drücken der Tastenkombination *STRG-Leertaste*, wenn der Cursor hinter dem gesuchten Wort oder Parameter steht.

Einige wichtige Grafikmethoden sind:

| | |
|---|---|
| *clearRect(int,int,int,int)* | löscht ein Rechteck |
| *drawRect(int,int,int,int)* | zeichnet ein Rec*hteck* |
| *fillRect(int,int,int,int)* | zeichnet ein gefülltes Rechteck |
| *drawOval(int,int,int,int)* | zeichnet eine Ellipse |
| *fillOval(int,int,int,int)* | zeichnet eine gefüllte Ellipse |
| *drawArc(int,int,int,int,int,int)* | zeichnet einen Kreisb*ogen* |
| *fillArc(int,int,int,int,int,int)* | zeichnet einen gefüllten Kreisbo*gen* |
| *drawString(string,int,int)* | zeichnet einen Text |
| *setColor(Color)* | setzt die Zeichenfarbe |
| *setXORMode(Color)* | setzt den invertierenden Zeichenmodus |
| *setPaintMode(Color)* | setzt den „normalen“ Zeichenmodus |

Zahlreiche weitere findet man im Hilfesystem.

Die Anzahl der zur Verfügung stehenden Punkte (*Pixel*) in den beiden Richtungen ergibt sich aus der momentanen Größe der Komponenten (wobei durchaus auch außerhalb dieses Bereichs gezeichnet werden kann). Arbeiten wir z. B. mit einem Applet, dann erhalten wir die aktuelle Breite durch Aufrufen der Methode *getWidth()* und die Höhe durch *getHeight()*. Der Ursprung des Koordinatensystems – der Punkt (0|0) – liegt in der oberen linken Ecke.

Die *Zeichenfarben* werden durch Java-Konstante gesetzt, die wir von der Klasse *Color* erhalten. Beispiele sind *Color.WHITE, Color.RED* usw. (Java-Konstante schreibt man groß!) Andere Farben als die vordefinierten erhält man durch Angabe der RGB-Werte wie in vorigen Abschnitt angegeben.

Im Bereich der Computergrafik ist der *Zeichenmodus* von Bedeutung, um bei Bedarf gezeichnete Elemente wieder löschen zu können, ohne den Hintergrund zu verändern. Im invertierenden *XOR-Mode* wird die Farbe der veränderten Punkte „umgedreht“, aus Weiß wird Schwarz usw. Zweimaliges Zeichnen bedeutet dann zweimaliges Umdrehen: der alte Zustand ist wieder hergestellt. In Java wird der Zeichenmodus durch Aufruf der beiden letzten der oben angegebenen Methoden des Graphics-Objekts eingestellt.

Bevor wir zu einem konkreten Beispiel kommen, müssen wird noch die Konventionen für *Java-Namen* kennen lernen.

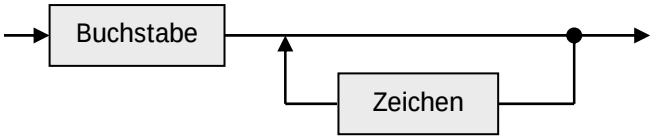
## 3.2 Bezeichner („Namen“) in Java-Programmen

In Java erhalten alle wesentlichen Elemente eines Programms eigene Namen.

- Einige Namen, die *reservierten Worte*, sind vorgegeben und dürfen nicht neu vergeben werden: z. B. *class, int, void, ...* Aus diesen Worten werden die leeren Programme gebildet, die von der Entwicklungsumgebung automatisch erzeugt werden. Zusätzlich werden darin Begriffe benutzt, die in den Bibliotheken definiert wurden – z. B. *Applet* oder sichtbare Komponenten wie *Button*. Auch diese Namen sollten auf keinen Fall umdefiniert werden.
- Für neu eingeführte Größen müssen selbst gewählte Namen gefunden werden. Dafür gelten die folgenden Regeln:
  - Alle Zeichen sind signifikant. Bezeichner dürfen also beliebig lang sein.
  - Das erste Zeichen muss ein Buchstabe[6] sein. Außer bei Klassennamen sollten Namen mit einem kleinen Buchstaben beginnen – müssen es aber nicht.
  - Leerzeichen sind verboten, nationale Sonderzeichen (wie Umlaute) aber erlaubt.
  - Groß- und Kleinschreibung wird unterschieden.
  - Gültige Bezeichner sind z. B. *a, test, einErsterVersuch, Applet12, . . .*
  - Erlaubte Namen erhält man, wenn der Name einem möglichen Durchlauf im folgenden *Syntaxdiagramm* entspricht:

Syntaxdiagramme bestehen aus *Graphen*, also *Kanten* (Pfeilen) und *Knoten* (Kästchen bzw. Ellipsoiden), die Teile der Programmiersprachensyntax beschreiben. Ein syntaktisch richtiges Programmstück stellt einen möglichen Weg durch den Graphen dar. Fehlerhafte Programme führen in Sackgassen. Trifft man auf dem Weg auf ein Ellipsoid, dann stellt dieses einen *Schlüsselbegriff* der Programmiersprache dar, der nicht weiter ersetzt werden kann. Kästchen enthalten Sprachelemente, die in anderen Syntaxdiagrammen noch weiter zu präzisieren sind.

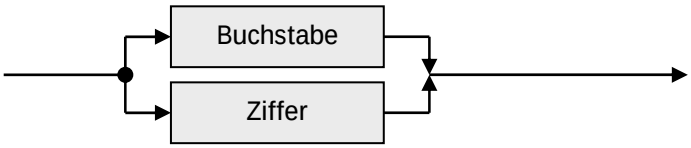
**Bezeichner:**



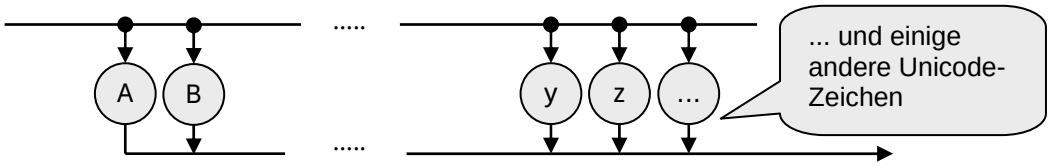

Das Diagramm ist so zu lesen: Ein *Bezeichner* beginnt mit einem *Buchstaben*. Ggf. werden weitere *Zeichen* angehängt, wobei noch offen ist, was unter Buchstaben und Zeichen zu verstehen ist. Buchstaben und Zeichen müssen noch definiert werden.

[6] Es dürfen auch einige Sonderzeichen wie der Unterstrich oder das Dollarzeichen dort stehen. Davon sollte man aber nur sparsam Gebrauch machen.

**Zeichen:**




**Buchstabe**:




**Ziffer:**


Beispiele:

- für korrekte Bezeichner: *a, a1, betragIn$, meinErstesProgramm, ...*
- für falsche Bezeichner: *1, 1A, klammer@ffe, mein erstes Programm, ...*

Man vermeidet Namenskonflikte mit den für Java reservierten Bezeichnern weitgehend, wenn man deutsche Bezeichnungen wählt – weil Java englische benutzt[7].

Im selben *Namensbereich* (oder *Gültigkeitsbereich*) darf kein Bezeichner doppelt auftreten. Namensbereiche werden z. B. durch eine Klasse, eine Methode oder einen Block definiert – die Anweisungen, die von geschweiften Klammern eingeschlossen werden. Innerhalb solch eines Bereichs gibt es zwei Arten von Bezeichnern:

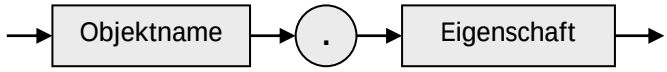
- *Lokale Namen* werden innerhalb eines Bereichs definiert, z. B. für Variable.
- *Globale Namen* wurden außerhalb des Bereichs definiert, in dem sie benutzt werden. So ist z. B. eine Variable der Klasse global innerhalb der Methoden oder eine Variable der Methode global für einen Schleifenblock.

Werden lokale Namen vergeben, die schon global vereinbart waren, dann *überschreiben* die lokalen Bezeichner die globalen, ersetzen diese also[8]. Alle Variablen existieren nur solange, wie der Programmteil, in dem sie vereinbart wurden, aktiv ist.

[7] Mit wenigen Ausnahmen: *name* z. B.
[8] Wenn die Ersetzung in dem entsprechenden Programmteil zulässig ist.

In Objekten werden Methoden und Eigenschaften des Objekts angesprochen, indem man der Methode oder Eigenschaft den Objektnamen – durch einen Punkt getrennt – voranstellt.

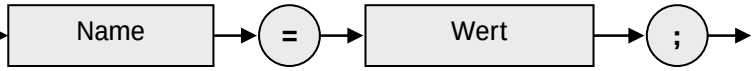
**Objekteigenschaft:**




Da Objekte wieder Objekte enthalten können, und diese wieder Objekte usw., können Namen recht lang werden.

Der *Zuweisungsoperator* „=“ weist einer Eigenschaft (oder einer Variablen) den rechts angegebenen Wert zu. Er ist zu lesen als „... *erhält den Wert* ...“.

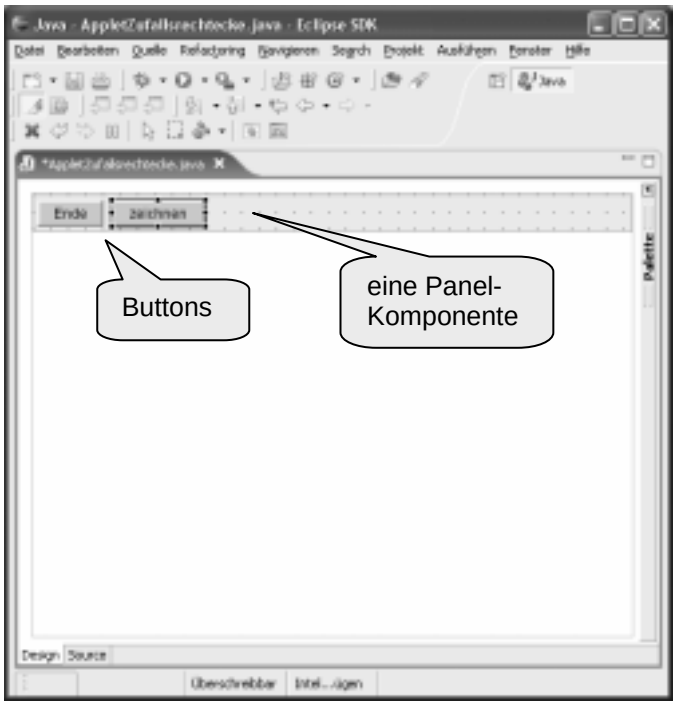
**Wertzuweisung:**




## 3.3 Zufallsgrafik

Wir wollen als Erstes ein Programm entwickeln, mit dem Rechtecke auf der Zeichenfläche erzeugt werden, deren Eckpunkte zufällig verteilt sind.

Dazu benötigen wir einen Button namens *bZeichnen*, der ein Zufallsrechteck erzeugt, wenn er angeklickt wird. Wir müssen also die Zeichenbefehle in der Ereignisbehandlungsmethode *actionPerformed* des *Zeichnen*-Buttons unterbringen. Diese wird in Eclipse automatisch erzeugt, wenn wir den *Zeichnen*-Button mit der rechten Maustaste anklicken und sie im Menüpunkte „Ereignisse“ auswählen. Zusätzlich erzeugen wir den schon bekannten *Ende*-Button zum Abbruch des Programms. Beide Buttons (und später weitere) wollen wir in einem Container unterbringen, der diese Knopfleiste übersichtlich zu-



sammenfasst. Für solche Zwecke ist die *Panel*-Komponente geeignet, deren *Layout*-Eigenschaft wir im Objektinspektor auf *null* setzen – ebenso wie die des Applets.

Für unsere Rechtecke bestimmen wir jeweils die obere linke Ecke, seine Breite und seine Höhe durch (geeignete) Zufallszahlen. Die erhalten wir durch eine nette Bitte an die *Math*-Klasse, uns solche zu liefern: *Math.random();* Die erhaltenen Zahlen liegen zwischen *0* und *1*. Wir „strecken" deshalb zuerst den Zahlenbereich mit einem Faktor *n* und erhalten Zufallszahlen zwischen *0* und *n*, danach bitten wir erneut die *Math*-Klasse, das Ergebnis auf eine ganze Zahl zu runden: *Math.round(Math.random()*n);*

Da das Ergebnis vom Typ *long* ist, der größere Zahlen speichert kann als *int*, erzwingen wir eine entsprechende *Typwandlung* (*type-casting*), indem wir die Direktive *(int)* voranstellen.

```
(int) Math.round(Math.random()*n);
```

Wenn wir mehrmals Zufallszahlen dieser Art erzeugen wollen, dann stellen wir am besten eine entsprechende *Funktion* bereit, die solche Zahlen berechnet. Der Rückgabewert der Funktion ist *int*. Mit *return* definieren wir diesen Wert. Übergeben wird als *Parameter* eine ganze Zahl *n*, die den Bereich bestimmt, aus dem die Zufallszahlen stammen.

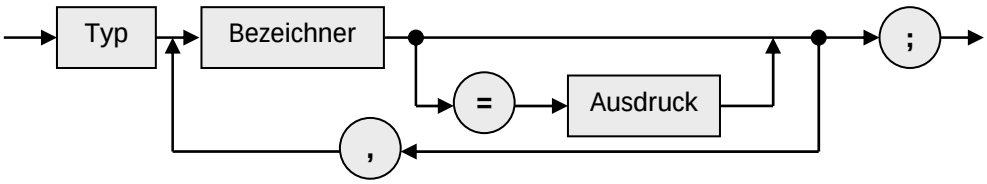
```
public int zz(int n)
{
  return (int) Math.round(Math.random()*n);
}
```

Variable müssen in Java zusammen mit ihrem Wert vereinbart werden, bevor man sie benutzen kann. Für unsere Rechtecke benötigen wir vier ganze Zahlen *x*, *y*, *b* und *h*, die die obere linke Ecke sowie Breite und Höhe des Rechtecks beschreiben. Bei Bedarf kann eine Variable gleich mit einem Wert initialisiert werden. In unserem Fall ist das für *x* eine Zufallszahl, die sich aus der Breite des Applets ergibt:

```
int x = zz(getWidth());
```

Allgemein muss eine Variablenvereinbarung der folgenden Syntax gehorchen:

**Variablenvereinbarung:**



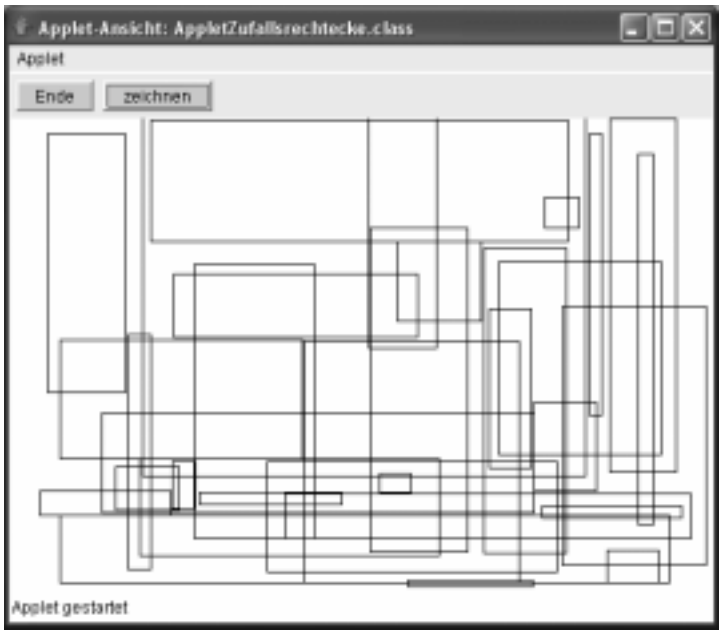

Als Datentypen stehen *Zahlen* (*int*, *double*, *long*, ...), *Wahrheitswerte* (*boolean*), *Zeichenketten* (*String*), *Farben* (*Color*) und weitere Klassen zur Verfügung. Die Vorgabewerte müssen natürlich dem Datentyp der Variablen entsprechen.

Mit diesen Kenntnissen ist es jetzt einfach, Zufallsrechtecke zu erzeugen. Wir

- vereinbaren Variable *x* und *y* für die linke obere Ecke des Rechtecks und initialisieren sie mit Zufallszahlen so, dass sie innerhalb der Appletfläche liegen. Bei *y* berücksichtigen wir noch die Höhe des Panels mit den Buttons, damit die Rechtecke nicht „unter“ diesem liegen.
- verfahren dann entsprechend für die Variablen *b* und *h*, die die Breite und Höhe des Rechtecks zufällig so wählen, dass dieses ganz im Zeichenfenster liegt. Dazu werden die vorher bestimmten Werte für *x* und *y* bei der Ziehung der Zufallszahlen mit berücksichtigt.
- ermitteln mithilfe der Funktion *getGraphics()* den aktuell zur Verfügung stehenden Grafikkontext
- und bitten dann diesen, das Rechteck mit den angegebenen Werten zu zeichnen.

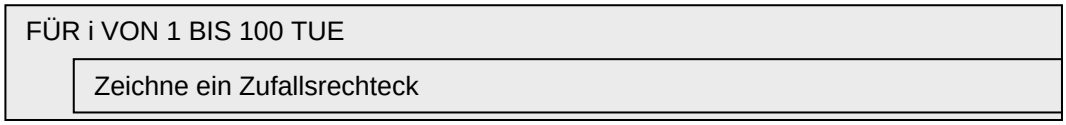
```
public void actionPerformed(java.awt.event.ActionEvent e)
{
  int x = zz(getWidth()-10)+5;
  int y = zz(getHeight()-42)+38;
  int b = zz(getWidth()-x-10)+5;
  int h = zz(getHeight()-y-10)+5;
  Graphics g = getGraphics();
  g.drawRect(x,y,b,h);
}
```

Klickt man den *Zeichnen*-Knopf mehrmals mit der Maus an, dann erhält man ein Bild ähnlich dem nebenstehenden.



## 3.4 Viele Rechtecke

Wenn man viele Rechtecke erzeugen will, ist das dauernde Anklicken des Zeichenbuttons etwas mühsam. Wir können diesen Vorgang etwas vereinfachen, wenn wir bei jedem Mausklick z. B. 100 Rechtecke zeichnen lassen. Dafür benutzen wir die aus dem letzten Abschnitt bekannte *Zählschleife*:

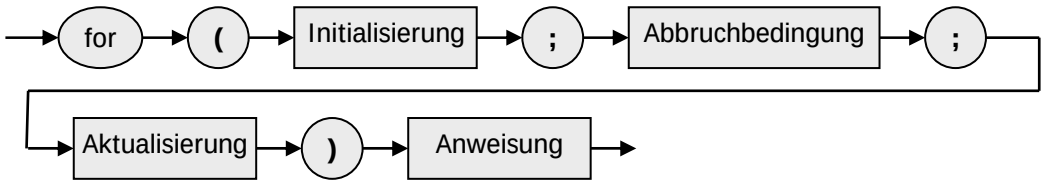



Die *Zählvariable i* benötigen wir zum Zählen der Durchläufe durch den Schleifenkörper, also zum Zählen der schon gezeichneten Rechtecke. In Java stellen wir den Rahmen dieser Schleife wie gehabt dar:

```
for (int i=0; i<100; i++)
{
  ... //Rechtecke zeichnen
}
```

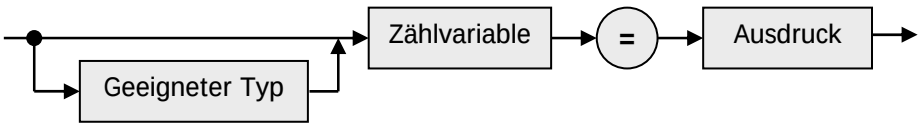
Allgemein muss eine Zählschleife in Java der folgenden Syntax gehorchen:

**Zählschleife:**

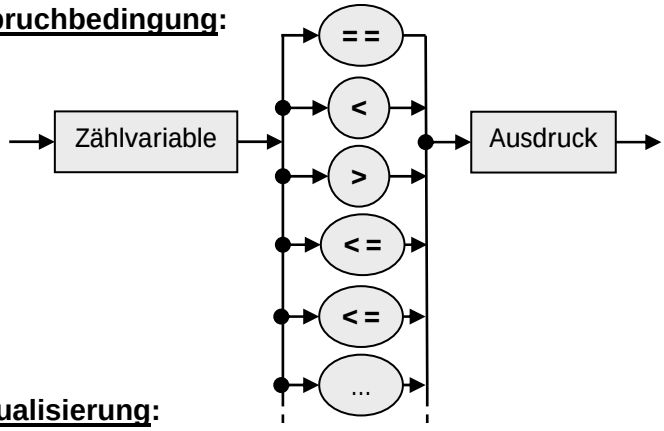



Wir benutzen in diesem Buch Zählschleifen nur in der schon angegebenen reduzierten Form. In dieser beutet *Initialisierung*, dass eine Zählvariable einen Anfangswert bekommt. Wurde sie noch nicht vereinbart, dann wird ihr der Datentyp (meist: *int*) vorangestellt. Damit ist sie für den folgenden Block gültig. Als *Abbruchbedingung* geben wir an, wann die Ausführung der Schleife beendet wird. Meist steht dort ein Ausdruck, der die Werte der Zählvariable begrenzt (z. B.: *i < 100*). Im *Aktualisierungsteil* wird der Wert der Zählvariablen bei jedem Durchlauf verändert. Meist wird er um Eins erhöht (*i++*).

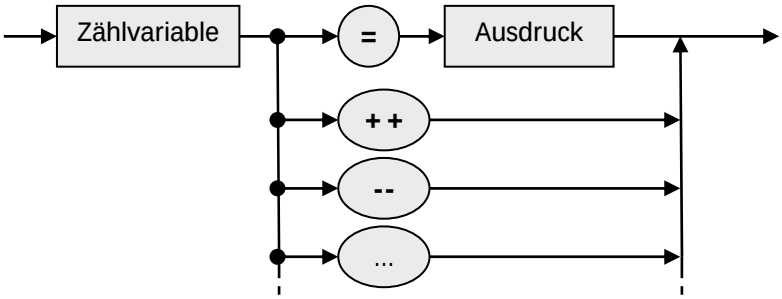
**Initialisierung:**



**Abbruchbedingung:**

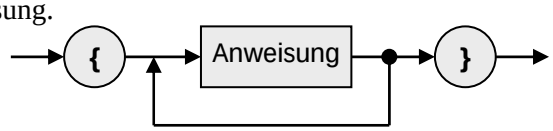



**Aktualisierung:**




Die einzelne wiederholte Anweisung besteht fast immer aus einer *Verbundanweisung*, also aus einem *Block*, der mehrere Anweisungen[9] zusammen klammert. Nur in Ausnahmefällen steht hier eine einzelne Anweisung.

**Verbundanweisung oder Block:**




In diesen Block fügen wir die schon bekannten Anweisungen zum Zeichnen von Zufallsrechtecken ein. Damit wir neue Effekte bekommen, zeichnen wir jetzt im *invertierenden Zeichenmodus* ausgefüllte Rechtecke. Dabei geben wir die Hintergrundfarbe, aus der sich das Ergebnis des XOR-Modes zusammen mit der Zeichenfarbe ergibt, als WEISS an.

```
g.setXORMode(Color.WHITE);
g.fillRect(x,y,b,h);
```

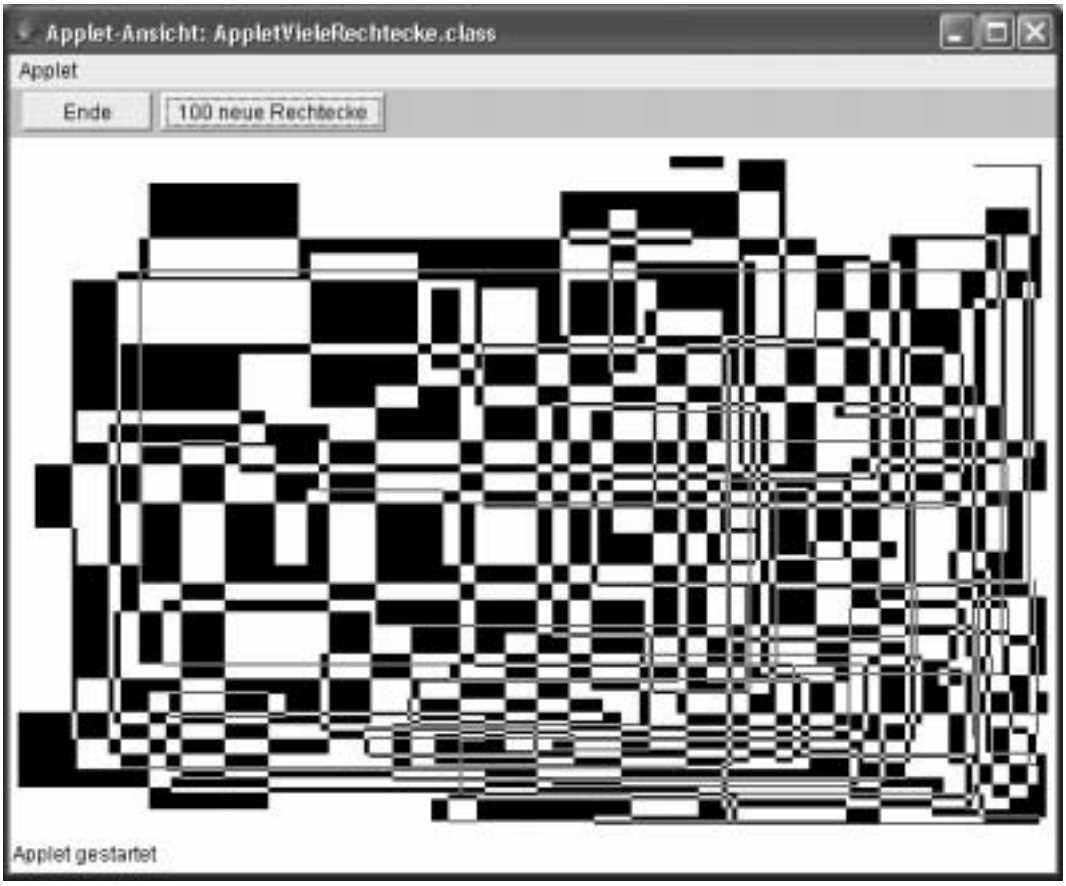
Werden zwei Rechtecke überlappend gezeichnet, dann dreht der zweite Zeichenvorgang die Farben des ersten wieder um: wie erhalten eine Art „Schachbrettmuster“.

[9] Viele der zur Verfügung stehenden Anweisungen findet man im Anhang. Meist handelt es sich um Methodenaufrufe oder Wertzuweisungen.

Die Reaktion des Zeichen-Buttons wird also wie folgt beschrieben:

```
public void actionPerformed(java.awt.event.ActionEvent e)
{
  for (int i=0; i<100; i++)
  {
    int x = zz(getWidth()-10)+5;
    int y = zz(getHeight()-42)+38;
    int b = zz(getWidth()-x-10)+5;
    int h = zz(getHeight()-y-10)+5;
    Graphics g = getGraphics();
    g.setXORMode(Color.WHITE);
    g.fillRect(x,y,b,h);
  }
}
```

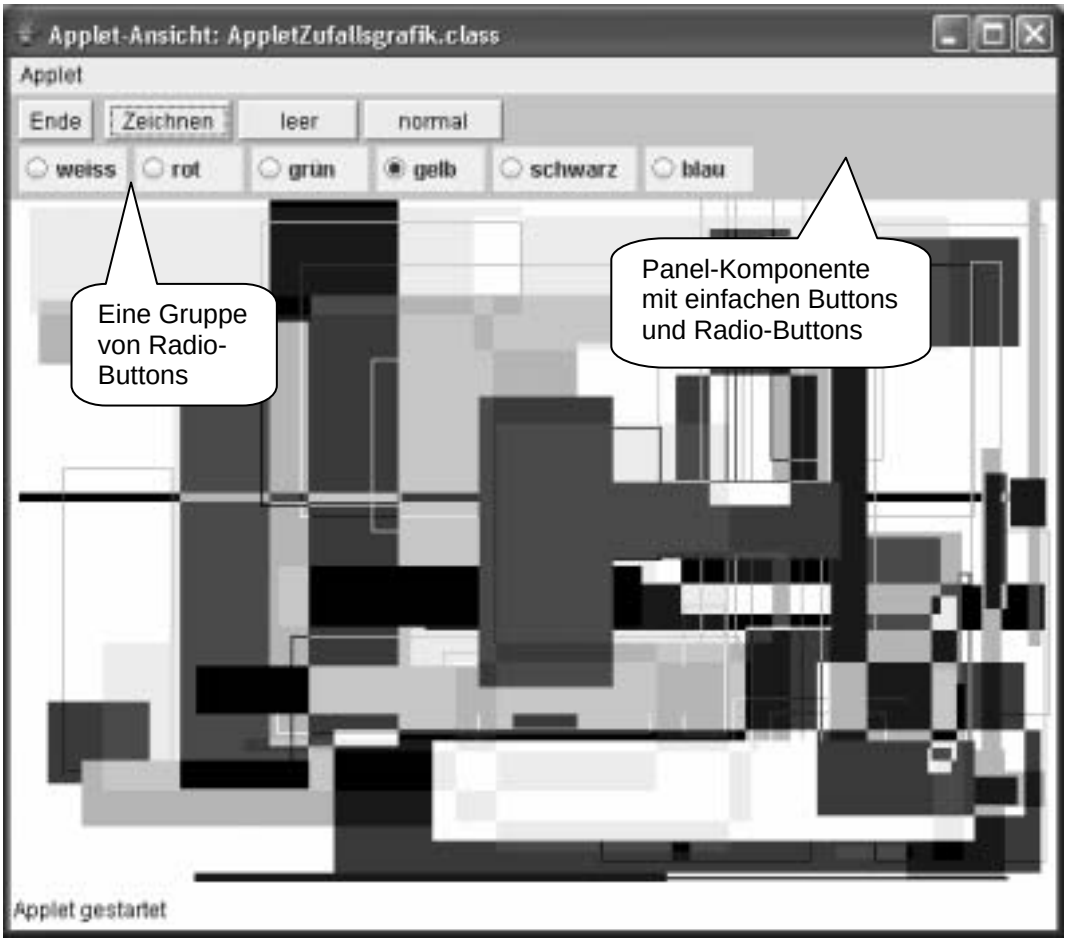
die beiden neuen Anweisungen



Wir wollen jetzt unser Programm etwas ausbauen, also um Möglichkeiten anreichern,

- die Zeichenfarbe zu wechseln,
- wahlweise die Rechteckflächen zu füllen - oder nicht
- und die Rechtecke entweder invertierend oder überschreibend zu zeichnen.

Dazu ergänzen wir die vorhandene Form um zwei Buttons *bGefuellt* und *bInvertierend* sowie einige Swing-Radiobuttons, deren Namen (*rbGruen, …*) und Text-Eigenschaften wir aus den Farben ableiten.




Nach dem Hinzufügen der üblichen Ereignisbehandlungsmethoden (*actionPerformed*) müssen wir nun entscheiden, wie das Programm gesteuert werden soll.

Zuerst einmal müssen wir dafür sorgen, dass von den Radiobuttons immer nur einer aktiv ist. Beim Schwarz-Knopf (*rbSchwarz*) setzen wir die *selected*-Eigenschaft auf *true*, weil anfangs schwarz gezeichnet werden soll. Übersetzen wir das Programm, dann können wir alle Radiobuttons unabhängig von einander bedienen – sie können also alle gleichzeitig aktiv sein. Um das zu verhindern, werden wir sie zu einer *Knopfgruppe* zusammenfassen, die wir *farbgruppe* nennen, und das erfordert etwas Handarbeit.

An der Stelle im Programm, wo die Komponenten erzeugt werden (das hängt etwas vom Entwicklungssystem ab: in Eclipse z. B. in einer Methode *getPanel()*), vereinbaren wir eine *ButtonGroup*:

```
ButtonGroup farbgruppe = new ButtonGroup();
```

Zu dieser fügen wir dann alle benutzten Radiobuttons hinzu. (Natürlich müssen die dann schon vorhanden sein!)

```
farbgruppe.add(rbWeiss);
farbgruppe.add(rbRot);
farbgruppe.add(rbGruen);
farbgruppe.add(rbGelb);
farbgruppe.add(rbSchwarz);
farbgruppe.add(rbBlau);
```

Jetzt ist immer nur einer der der Farbknöpfe aktiv. Insgesamt haben wir nun eine Reihe von Auswahlknöpfen – aber wie benutzt man sie?

Jeder Button soll eine Zeicheneigenschaft (Farbe, Zeichenmodus, ..) bei Bedarf auf einen vorgegebenen Wert setzen. Nach dieser Aktion soll diese Eigenschaft solange unverändert bleiben, bis sie wieder bewusst gewechselt wird. Wir benötigen deshalb *Variable*, in denen die Zeicheneigenschaften gespeichert werden. Da diese Größen während des gesamten Programmlaufs ihre Werte behalten müssen, dürfen sie nicht innerhalb einer Ereignisbehandlungsmethode vereinbart sein, sondern gehören als *globale Variable* direkt in den Hauptteil des Programms, am besten direkt an den Anfang:

```
Color farbe = Color.BLACK;
boolean gefuellt = false;
boolean imXORMode = false;
```

Der Typ einer Variablen muss der Größe entsprechen, die gespeichert werden soll. Bei Farben handelt es sich um die AWT-Klasse *Color*, bei den anderen merken wir uns in einer *booleschen Variablen* (einem *Wahrheitswert*), die nur die Werte *true* und *false* annehmen kann, ob die zu zeichnenden Rechtecke gefüllt bzw. invertierend gezeichnet werden sollen - oder nicht. Immer, wenn ein Button angeklickt wird, muss die entsprechende Variable den richtigen Wert erhalten. Globale Variable erfordern meist das Setzen von Anfangswerten als Vorgaben, mit denen das Programm startet. Das kann gleich bei ihrer Vereinbarung geschehen.

Als Beispiel wollen wir die Reaktion des *rbBlau*-Buttons betrachten, wenn er angeklickt wird:

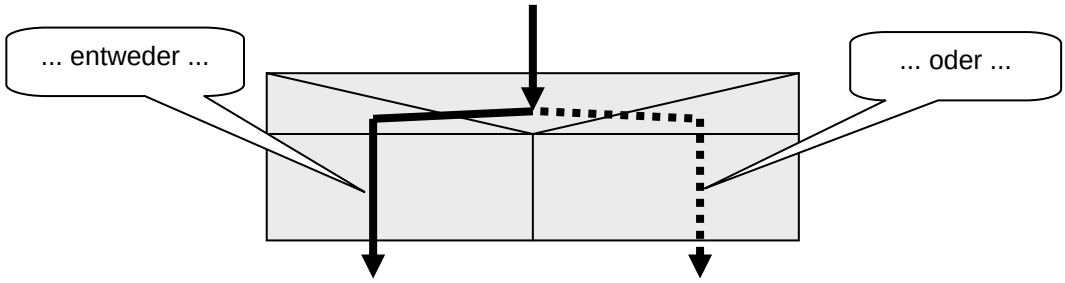
```
public void actionPerformed(java.awt.event.ActionEvent e)
{
  farbe = Color.BLUE;
}
```

Etwas komplizierter gestaltet sich die Angelegenheit bei den Buttons, die den Zeichenstil festlegen. Hier erhält die entsprechende Variable ja keinen festen Wert, sondern ihr Wert wird jedes Mal „umgedreht“. Dabei sollte auch gleich die Aufschrift des Buttons geändert werden, da er ja bei jedem Mausklick seine Funktion ändert. Wir haben es also mit einer *Alternative* oder *Fallunterscheidung* zu tun: je nach augenblicklicher Einstellung werden Variable und Aufschrift auf einen anderen Wert gesetzt. Der aktuelle Zustand ergibt sich aus den Werten der booleschen Variablen.

| gefuellt: wahr | gefuellt: falsch |
|---|---|
| gefuellt ← false | gefuellt ← true |
| bGefuellt erhält die Aufschrift „gefüllt“ | bGefuellt erhält die Aufschrift „leer“ |

(Bei den Aufschriften der Buttons handelt es sich um *Zeichenfolgen* vom Datentyp *String*, die in Java durch „Gänsefüßchen“ eingeschlossen werden.)

Diese Darstellung von Befehlsfolgen in Kastenform kennen wir schon als *Struktogramm*. Struktogramme werden von oben nach unten durchlaufen, beschreiben den Ablauf eines Programmstücks. In unserem Fall wird zuerst entschieden, welche Seite der Alternative ausgewählt wird. Danach werden die entsprechenden Befehle ausgeführt.

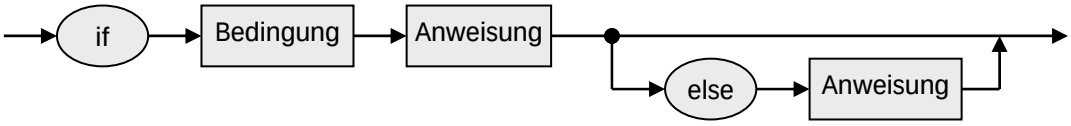


Die Übersetzung in Java lautet:

```
public void actionPerformed(java.awt.event.ActionEvent e)
{
  if(gefuellt)
  {
    bGefuellt.setLabel("gefüllt");
    gefuellt = false;
  }
  else
  {
    bGefuellt.setLabel("leer");
    gefuellt = true;
  }
}
```

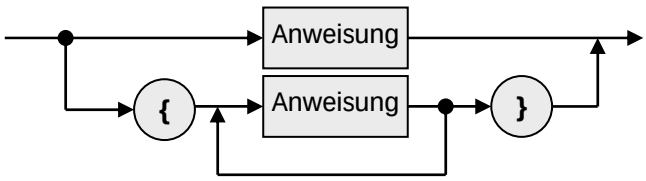
Der Aufbau einer Alternative wird durch ein Syntaxdiagramm beschrieben.

**Alternative:**



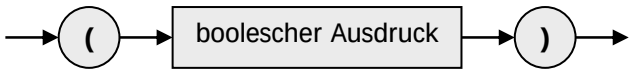

Der *else*-Zweig kann also bei Bedarf entfallen (*einseitige Alternative*), und die auszuführende Anweisung kann aus einer einzelnen, (meist aber) aus mehreren Anweisungen bestehen, die dann mit geschweiften Klammern zu einem Block zusammengefügt werden.

**Anweisung:**




Als *Bedingunge*n eignen sich alle Ausdrücke, die als Ergebnis entweder *wahr* (*true*) oder *falsch* (*false*) ergeben. Das sind z. B. die booleschen Variablen selbst, aber auch Vergleiche (*i < 100*) und andere.

**Bedingung:**



Zulässige Alternativen sind

- ```
  if(i > 0) i++;
  else i--;
  ```
- ```
  if(Farbe == Color.RED) g.drawRect(x,y,b,h);
  else g.drawOval(x,y,b,h);
  ```
- ```
  if(Name.equals("Meier"))
  {
    Vorname = "Hans";
    Alter   = 44;
  }
  ```
- ```
  if(gefunden) tfAusgabe.setText("gefunden!");
  ```

Wir brauchen jetzt nur noch vor dem Zeichnen eines Rechtecks dafür zu sorgen, dass die gewählten Einstellungen auch umgesetzt werden. Das geschieht in der *actionPerformed*-Methode des Buttons *bZeichnen*: (Nur die „neuen" Anweisungen sind fett gedruckt!)

```
public void actionPerformed(java.awt.event.ActionEvent e)
{
  int x = zz(getWidth()-10)+5;
  int y = zz(getHeight()-42)+38;
  int b = zz(getWidth()-x-10)+5;
  int h = zz(getHeight()-y-10)+5;
  Graphics g = getGraphics();
  g.setColor(farbe);
  if(imXORMode) g.setXORMode(Color.WHITE);
  else g.setPaintMode();
  if(gefuellt) g.fillRect(x,y,b,h);
  else g.drawRect(x,y,b,h);
}
```

Je nach Variablenwert invertierend oder normal zeichnen.

entsprechend für „gefuellt"