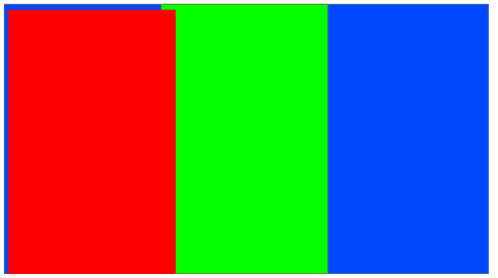
RGB-Farben



RGB : rot – grün – blau
In Processing kann sowohl der Hexidezimalcode (sechs Ziffern jeweils aus {0; 1; ... ;9; A; B; C; D; E; F}) als auch die dezimale Darstellung (drei Zahlen zwischen 0 und 255 mit Komma getrennt) zur Datstellung der Farbanteile verwendet werden.

Farbbeispiele:

schwarz :	0, 0, 0	hexadezimal 000000
gelb:	255, 255, 0	hexadezimal FFFF00
rot:	255, 0, 0	hexadezimal FF0000
grün:	0, 255, 0	hexadezimal 00FF00
blau:	0, 0, 255	hexadezimal 0000FF

in Processing:
background(255,255,0); oder background(#FFFF00);

Processing

Processing ist eine auf Java basierende objektorientierte Programmiersprache, die mit dem Hintergedanken entwickelt wurde, künstlerische Ideen schneller umzusetzen.

Beispiele:

Mandelbrotmenge (selbstähnliche Figur)

kleine Spiele wie snake, tetris, pacman, ...

Processing

Grundaufbau eines Processing-Programms:

Grundein-
stellungen

```
void setup()
{
      size(500,500);                  // legt die Größe
                                      // des Fensters fest
      background(100,100,100);  // legt die
                                            // Hintergrundfarbe fest
 }

void draw()
{
                .....
}
```

Processing

setup-Methode:

Mit dem Schlüsselwort **void** wird eine Methodendefinition eingeleitet. Die **setup-**Methode ist eine Besonderheit von Processing.
In ihrem Rumpf (d. h. in dem Programmteil zwischen den geschweiften Klammern) werden die Grundeinstellungen vorgenommen. Die **size-**Anweisung, die die Größe des Ausgabefensters festlegt, sollte hier als erste Anweisung stehen.

Bei Aufruf des Programms werden die Anweisungen im Rumpf der setup-Methode in der vorgegebenen Reihenfolge genau einmal durchgeführt. In unserem Beispielprogramm wird bei Programmstart ein Fenster der Größe 500x500 geöffnet und die Hintergrundfarbe auf den entsprechenden Wert gesetzt.

Processing

Anweisungen:

Es gibt verschiedene Arten von Anweisungen, eine davon haben wir in diesem Beispielprogramm schon kennengelernt: Methodenaufrufe wie z.B.

```
size(500,500);
background(100,100,100);
```

Jeder Methodenaufruf beginnt mit dem Methodennamen, gefolgt von einer öffnenden runden Klammer und einer Folge von Argumenten, die durch Kommas getrennt sind. Die schließende runde Klammer und der Strichpunkt beenden den Methodenaufruf.

Syntax: Methodenname (Argument1, ... , Argumentn) ;

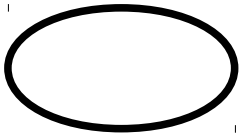
Die Liste der Argumente kann auch leer sein.

Exkurs: Zeichenmethoden

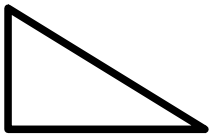
In Processing gibt es sehr viele vordefinierte Methoden.
Einfache Zeichenmethoden:

point(x, y)
   zeichnet einen Punkt an der Position (x,y)
line(x1, y1, x2, y2)
   zeichnet eine Strecke von (x1,y1) nach (x2,y2)

ellipse(x1,y1,breite,hoehe)
   zeichnet eine Ellipse mit dem Mittelpunkt (x1,y1) und der Breite breite und der Höhe hoehe

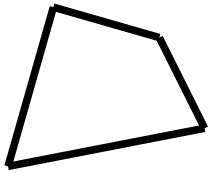


triangle(x1,y1,x2,y2,x3,y3)
   zeichnet ein Dreieck mit den Eckpunkten (x1,y1), (x2,y2), (x3,y3)



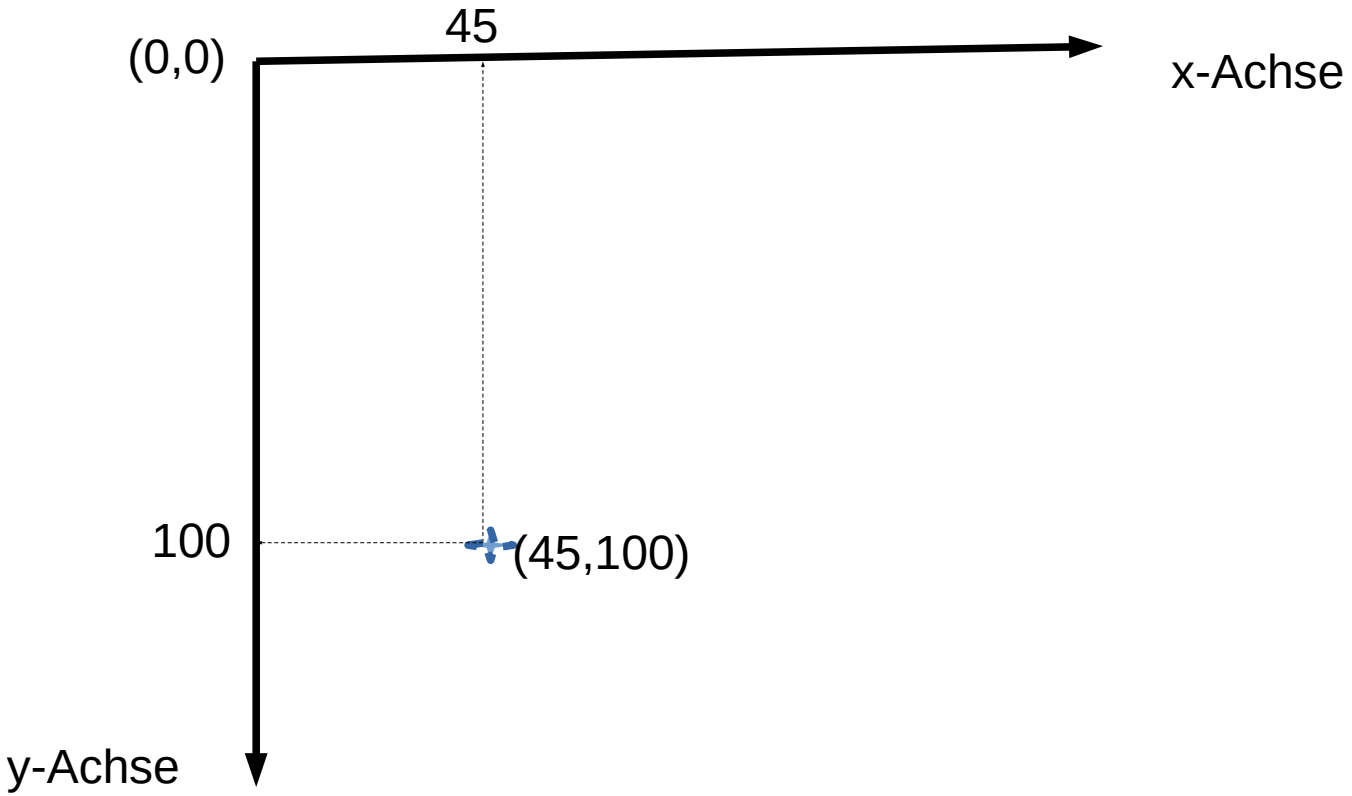
rect(x1,y1,breite, hoehe)
   zeichnet ein Rechteck mit (x1,y1) als linker oberer Eckpunkt mit der Breite breite und der Hoehe hoehe

quad(x1,y1,x2,y2,x3,y3,x4,y4)
   zeichnet ein Viereck mit den angegebenen vier Eckpunkten



Koordinatensystem

In der Informatik sieht ein Koordinatensystem etwas anders aus als in der Mathematik. Negative Koordinaten zur Positionierung gibt es nicht, der Nullpunkt ist in der linken oberen Ecke, die positive y-Achse zeigt nach unten.



Exkurs: Farbe, Strichdicke, Transparenz

Mit folgenden Befehlen kann man die Farbe, Strichdicke der Umrandung und die Transparenz von Objekten beeinflussen:

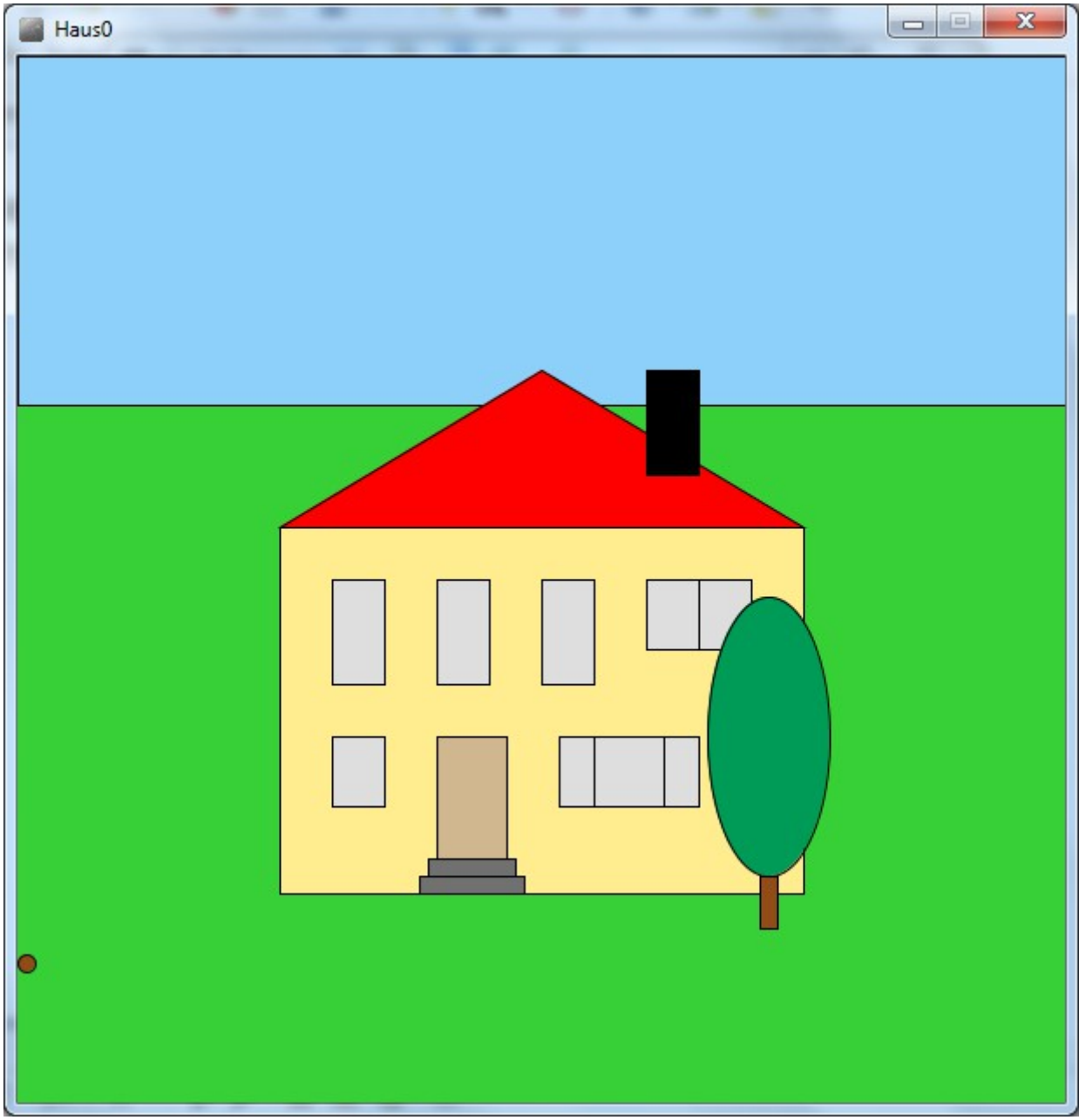
stroke(r,g,b)
    setzt die Farbe der Umrandung auf den RGB-Wert
noStroke()
    verhindert eine Umrandung
strokeWeight(zahl)
    setzt die Strichdicke auf zahl (in Pixel)
fill(r,g,b)
    setzt die Füllfarbe des Objekts auf den RGB-Wert
noFill()
    verhindert eine Füllung

Animation – Variablen und Konstanten

Für unsere Zeichenmethoden im Programm haben wir bisher feste Zahlen (Konstanten) verwendet. Ihr Wert ist über den Zahlenwert direkt angegeben.

Unser Haus wird immer an derselben Stelle gezeichnet, das Bild verändert sich nicht.

Beispiel aus den Vorlagenverzeichnis: Projekt Haus0



Animation des Balls

Wir wollen den Ball am linken Rand des Bildes von links nach rechts rollen lassen.

aktuell: ellipse(5,520,10,10);

Um den Ball von links nach rechts zu bewegen, müssen wir ihn immer wieder an einer neuen Position zeichnen, d.h. die x-Koordinate muss sich ändern.

Einführen eines Platzhalters für die x-Koordinate **(Variable)**

Variablen

Eine Variable repräsentiert einen Speicherplatz im Rechner, in dem man Werte speichern und auslesen kann.

In Processing (und Java) muss für jede Variable **vor ihrer ersten Verwendung** festgelegt werden, welchen **Datentyp** sie hat.

Mögliche Datentypen (u.a.):

int	ganze Zahlen (integer), z.B. -4; 6; 100
float, double	Gleitkommazahlen, z.B. 2.4, -0.01
char	Zeichen (character), z.B. 'a'; '-'; '+'; 'q'
String	Zeichenketten, z.B. “das“, “1234“; “wir sind“

Vereinbarung

Regeln für die Benennung von Variablen, Methoden, Attributen, …:

In den Bezeichnungen sollten keine Leerzeichen, keine Umlaute und keine Sonderzeichen auftreten. Bindestriche und Unterstriche können verwendet werden.

Als Bezeichnung für Variablen sollten aussagekräftige Namen verwendet werden.

Für die x-Koordinate des Balls verwenden wir z.B. xKoord.

Deklaration

Um die Variable xKoord zu deklarieren, wird der passende Datentyp gesucht. Da die Koordinaten ganzzahlig sind, wählen wir als Datentyp **int**.

Die Deklaration der Variablen nehmen wir vor der setup()-Methode vor, indem wir die Typdeklaration

```
int xKoord;
```

in unser Programm schreiben. Damit haben wir festgelegt, dass in xKoord nur ganzzahlige Werte gespeichert werden können.

Aktuelles Programm:

```
int xKoord;

void setup()
{
     ...
}

void draw()
{
     ...
}
```

Initialisierung

Wenn wir jetzt xKoord verwenden, dann wissen wir nicht, welcher Wert in dieser Variablen gespeichert ist!

WICHTIG:
Der Anfangswert der Variablen muss festgelegt werden. Man sagt auch, die Variable muss **initialisiert** werden.

Der passende Ort für eine Initialisierung ist in diesem Fall die setup()-Methode.

Aktuelles Programm:

```
int xKoord;

void setup()
{
    xKoord = 5;    // halbe Breite
    ...
    ellipse(xKoord, 520,10,10);
    ...
}

void draw()
{
    ...
}
```

Wenn wir das Programm starten, dann sieht die Ausgabe erwartungsgemäß genauso aus wie vorher.

Wir müssen dafür sorgen, dass sich der Wert der Variablen xKoord ändert.

Durch die Anweisung

```
xKoord = xKoord + 1;
```

wird der Wert, der in xKoord gespeichert ist, um 1 erhöht und wieder in xKoord gespeichert.

Das ist *keine* mathematische Gleichung, sondern eine sogenannte Wertzuweisung und muss von rechts nach links gelesen werden.

Diese Wertzuweisung müssten wir laufend wiederholen, um den Ball von links nach rechts zu bewegen. In einer anderen Programmiersprache bräuchten wir hier noch einiges mehr.

Besonderheit von Processing:
Alles, was in der draw()-Methode steht, wird solange wiederholt, bis das Programm beendet wird.

Animation des Balls:
Das Zeichnen des Balls und das Erhöhen des Werts für die x-Koordinate muss in die draw()-Methode.

Aktuelles Programm:

```
int xKoord;

void setup()
{
    xKoord = 5;    // halbe Breite
    ...
}

void draw()
{
    ellipse(xKoord, 520,10,10);
    xKoord = xKoord + 1;
}
```

Jetzt bewegt sich etwas :-)

Arbeitsauftrag:
1. Ergänzen Sie Ihr Programm um die nötigen besprochenen Änderungen.

2. Überlegen Sie sich, wie sie erreichen können, dass Sie nur den Ball sehen, nicht aber alle alten Zeichnungen. Wenn Sie das Problem erkannt haben, beheben Sie es (wenn möglich).

3. Lassen Sie weitere Objekte durch das Bild rollen, fallen oder fliegen (z.B. Rauch aus dem Schornstein). Beschränken Sie sich auf einfache Objekte.

*4. Auf der rechten Seite einer Wertzuweisung können beliebige Rechenterme stehen. Die Modulo-Operation (Operator %) liefert den ganzzahligen Rest einer Division. Versuchen Sie damit, den Ball immer wieder von links nach rechts rollen zu lassen.

Exkurs: Bilder (1)

Wir haben viele Zeichenmethoden kennengelernt, mit denen wir Bilder zeichnen lassen können. Mit geeigneten Mitteln können wir die Einzelteile eines Bildes ansprechen und z.B. einfärben, bewegen, vergrößern, ...

Gelegentlich wollen wir einfach nur einen festen Hintergrund, der sich nicht verändert. Dazu eignen sich fertige jpg- oder png-Dateien.

Vorgehen zum Erstellen eines jpg- Hintergrundbilds:
Im Projektordner muss ein Verzeichnis `data` erstellt werden, in den die benötigte Bilddatei z.B. `all.jpg` hinein kopiert wird.
Um diese Bilddatei verwenden zu können, muss im Programm folgende Zeile (am besten als erste Programmzeile) verwendet werden:

```
/* @pjs preload="all.png"; */
```

Exkurs: Bilder (2)

Durch die beschriebene Programmzeile (! in Kommentarklammern !) kann das Bild geladen werden. Jetzt müssen wir es nur noch in das Programm einbauen. Dazu brauchen wir den vordefinierten Datentyp `PImage`.

Wir deklarieren eine Variable von diesem Datentyp z.B. mit

```
PImage bg;
```

In der setup-Methode bekommt diese Variable ihr Bild zugewiesen:

```
bg = loadImage("all.jpg");
```

Ab jetzt können wir mit Hilfe der Anweisung

```
background(bg);
```

dieses Bild als Hintergrundbild verwenden.

Objektorientierte Modellierung – Objektorientierte Programmierung

Nachteile unseres aktuellen Programms:
- unübersichtlich
- wenn wir mehrere Bäume, Häuser, ... haben wollen, dann wird es noch länger/unübersichtlicher
- thematische Gliederung nur rudimentär vorhanden
- bisher nur eine Aneinanderreihung von Anweisungen
- unflexibel für Änderungen (Haus soll 50 Pixel nach links)

Objektorientierte Modellierung – Objektorientierte Programmierung

1. Everything is an object, 2. Objects communicate by sending and receiving messages (in terms of objects), 3. Objects have their own memory (in terms of objects), 4. Every object is an instance of a class (which must be an object), 5. The class holds the shared behavior for its instances (in the form of objects in a program list), 6. To eval a program list, control is passed to the first object and the remainder is treated as its message

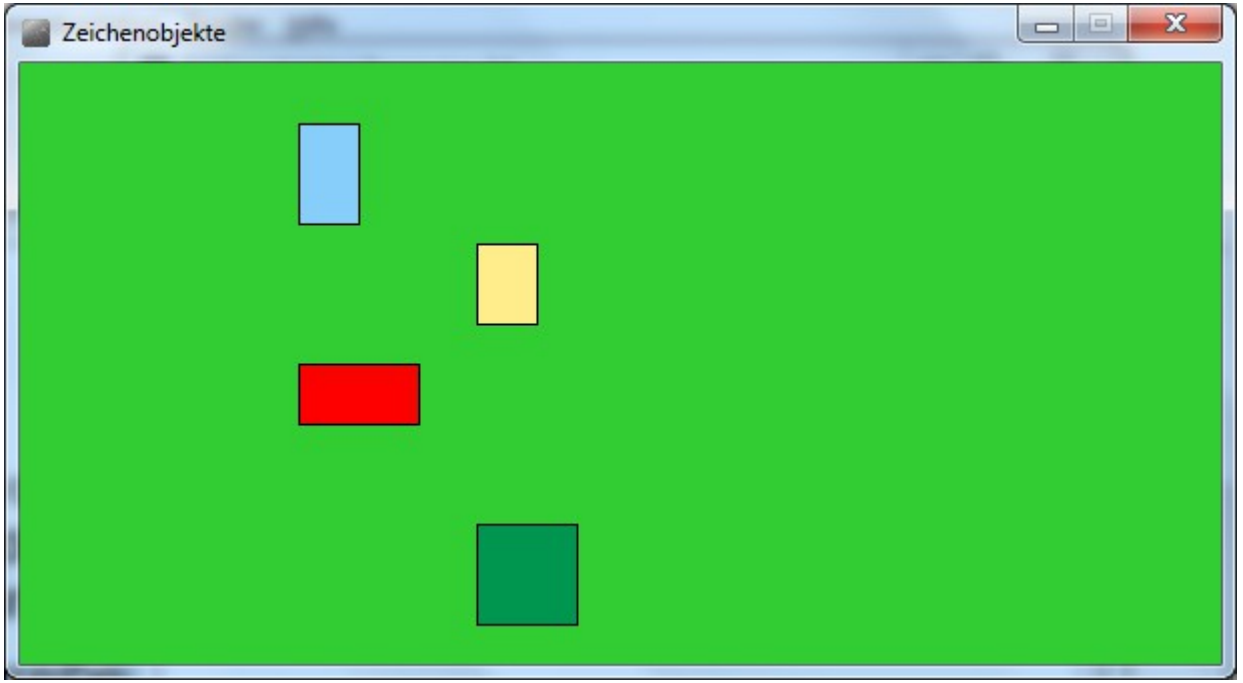
„1. Alles ist ein Objekt, 2. Objekte kommunizieren durch das Senden und Empfangen von Nachrichten (welche aus Objekten bestehen), 3. Objekte haben ihren eigenen Speicher (strukturiert als Objekte), 4. Jedes Objekt ist Instanz einer Klasse (welche ein Objekt sein muss), 5. Die Klasse beinhaltet das Verhalten aller ihrer Instanzen (in der Form von Objekten in einer Programmliste), 6. Um eine Programmliste auszuführen, wird die Ausführungskontrolle dem ersten Objekt gegeben und das Verbleibende als dessen Nachricht behandelt“

– Alan Kay: The Early History of Smalltalk (1993)

Rasende Rechtecke

Bevor wir uns an die objektorientierte Modellierung in Java machen, wollen wir die gelernten Grundkompetenzen noch einmal wiederholen.

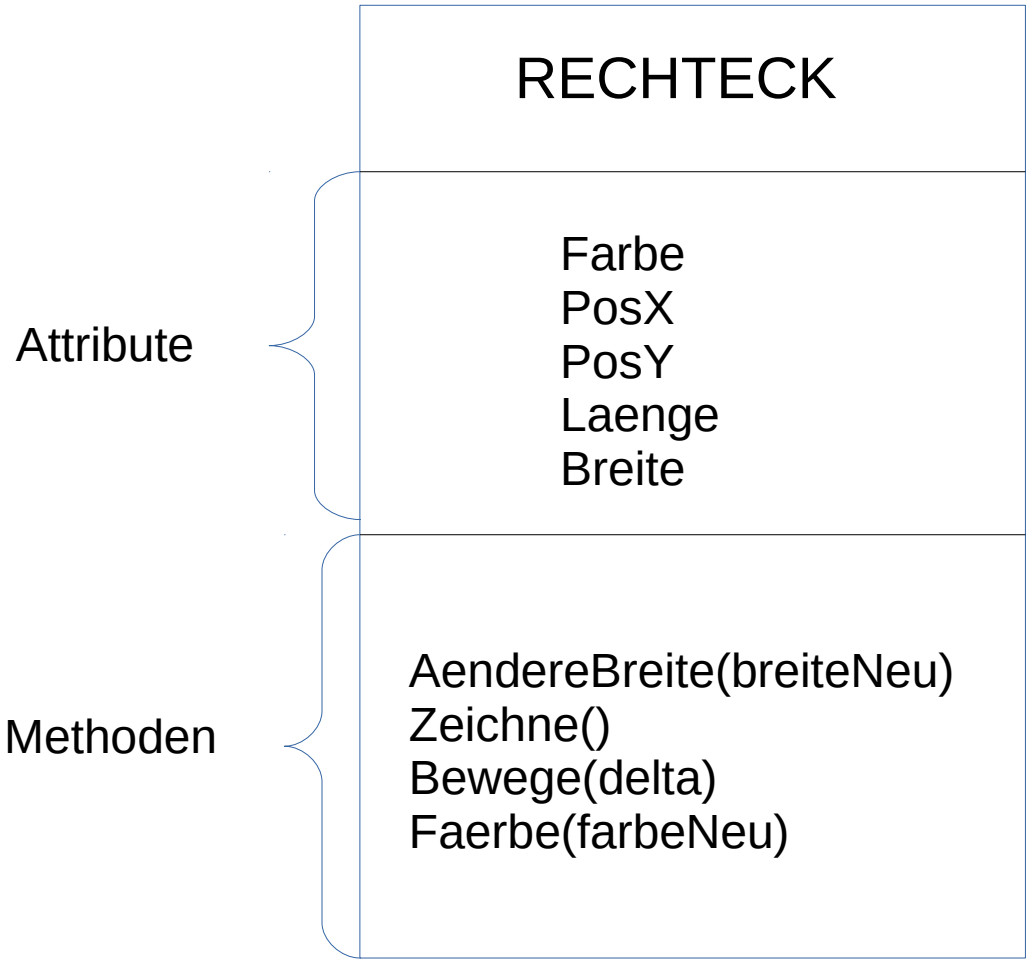
Schreiben Sie ein Programm in Java, das vier verschiedenfarbige Rechtecke in der Zeichenfläche darstellt (für die Schnellen: ... und zum Bewegen bringt).



Objektorientierte Modellierung

Hauptaufgabe der objektorientierten Modellierung ist es, die beteiligten Objekte zu erkennen, in Klassen zu beschreiben und die Interaktionen von Objekten unterschiedlicher Klassen zu ermöglichen.

Wiederholung: Klassendiagramm

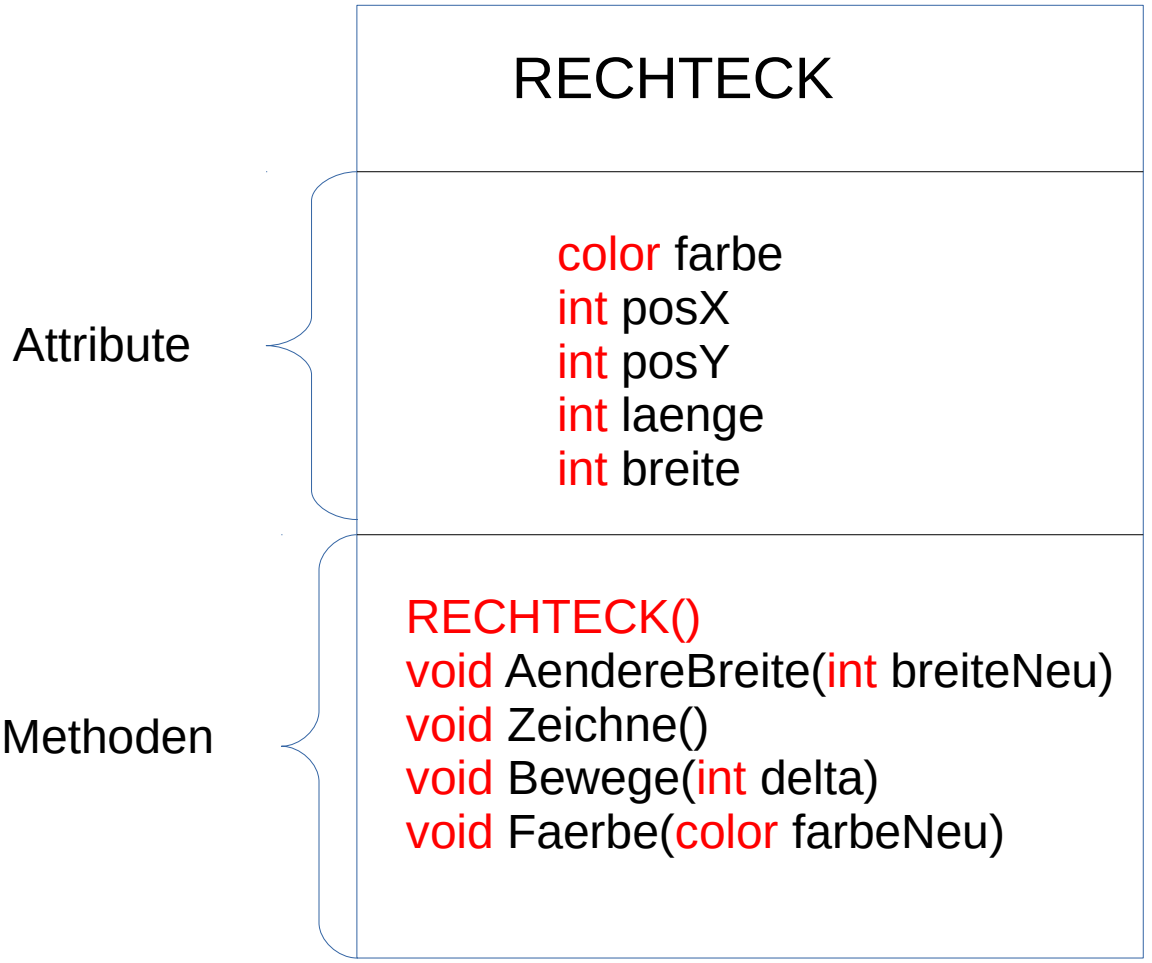


Ein Klassendiagramm dient als Bauplan für Objekte mit gleichen Eigenschaften und Methoden.

Ein konkretes Rechteck ist eine Instanz eines Klassendiagramms bzw ein Objekt der Klasse, die durch dieses Klassen-diagramm beschrieben ist.

```
int x1;
int x2;
int x3;
int x4;

void setup()
{
  size(600,300);
  background(50,205,50);   // Hintergrund

  x1 = 50;
  x2 = 50;
  x3 = 50;
  x4 = 50;

}

void draw()
{
  background(50,205,50);   // Hintergrund
  fill(135,206,250);      // blaue Farbe
  rect(x1,30,30,50);
  fill(255,236,139);     // hellgelbe Farbe
  rect(x2,90,30,40);
  fill(255,0,0);
  rect(x3,150,60,30);
  fill(0,150,80);
  rect(x4,230,50,50);
  x1 = (x1 +1)%600;
  x2 = (x2 + 2)%600;
  x3 = (x3 + 1)%600;
  x4 = (x4 + 2)%600;
}
```

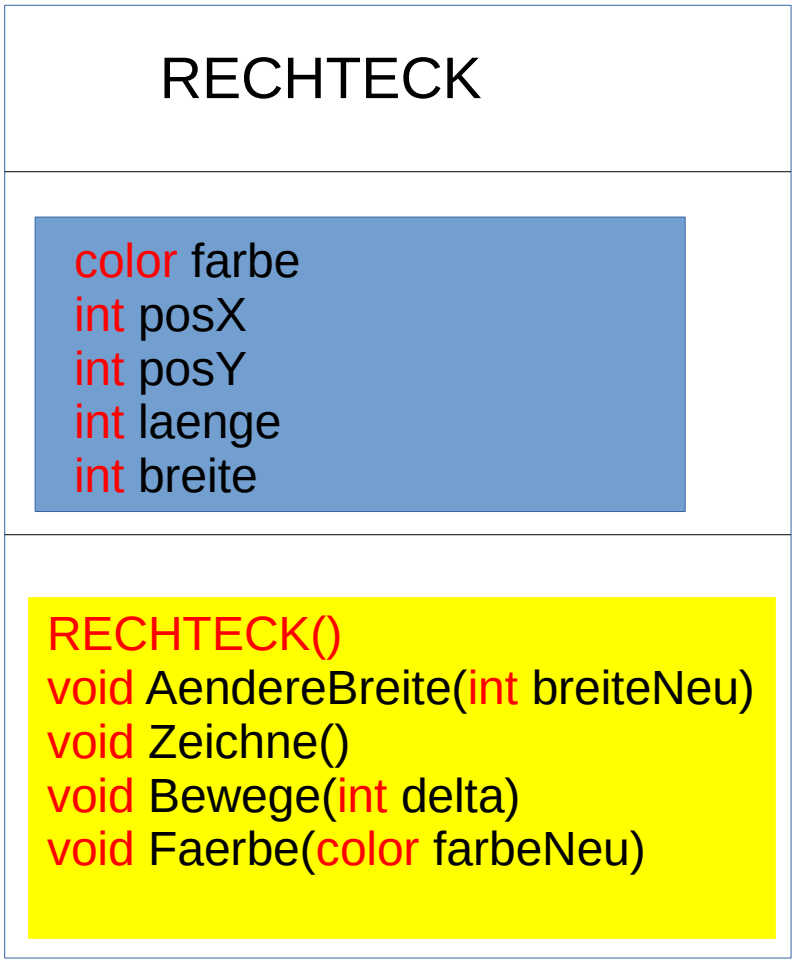
Erweitertes Klassendiagramm



Bei einem erweiterten Klassendiagramm werden die Attribute und die Methoden um die Angabe der Datentypen ergänzt.

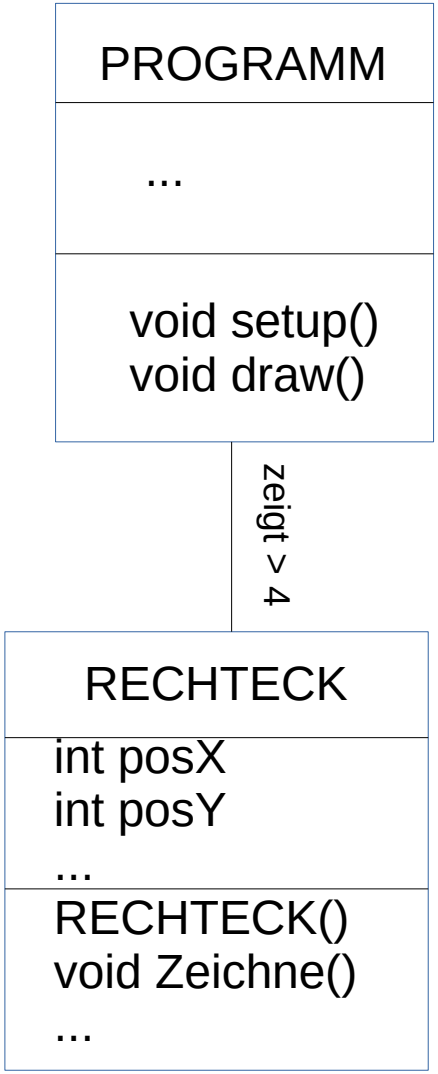
Methoden, die keinen Wert liefern, müssen mit void gekennzeichnet werden. Jede Klasse sollte einen Konstruktor (hier RECHTECK()) enthalten, der ein Objekt der Klasse erzeugen kann.

Vom erweiterten Klassendiagramm zum Programm




```
class RECHTECK
{
    color farbe;          // Deklaration, d.h.
    int posX;             // Festlegung des
    int posY;             // Datentyps der
    int laenge;           // Attribute
    int breite;

    RECHTECK()
    {
        farbe = #FFFFFF;   // Initialisierung
        posX = 50;         // d.h. Festlegung
        posY = 100;        // der Startwerte
        laenge = 60;       // fuer die Attribute
        breite = 60;
    }

    void Zeichne()
    {
        fill(farbe);
        rect(posX, posY, laenge, breite);
    }
}
```

Objektorientierte Modellierung
Rasende Rechtecke



erweitertes
Klassendiagramm

Im Hauptprogramm PROGRAMM muss die Beziehung zur Klasse RECHTECK umgesetzt werden. Dazu werden Referenzattribute benötigt.
Ein Referenzattribut speichert einen Verweis auf ein konkretes Objekt. Dazu muss ein Referenzattribut zuerst deklariert werden, d.h. sein Datentyp muss festgelegt werden.
In unserem Fall brauchen wir vier Referenzattribute des Datentyps RECHTECK, um die vier Rechtecke zu speichern. Geschrieben wird das vor der setup-Methode durch eine Anweisung wie

RECHTECK kasten1;

Im setup-Teil erzeugen wir die vier Objekte und weisen sie den Referenzattributen zu, indem wir folgende Anweisung verwenden:

kasten1 = new RECHTECK();

Nach dem Schlüsselwort new wird der Konstruktor RECHTECK aufgerufen und damit werden die Anweisungen ausgeführt, die in der Klasse RECHTECK im Rumpf des Konstruktors stehen.

Mit den Objekten wird durch den Aufruf von passenden Methoden kommuniziert. Wir haben z.B. die Methode Zeichne in die Klasse RECHTECK geschrieben. Um im draw-Teil des Hauptprogramms dieses Rechteck zu zeichnen, müssen wir die passende Methode aufrufen:

kasten1.Zeichne();

Wichtig:

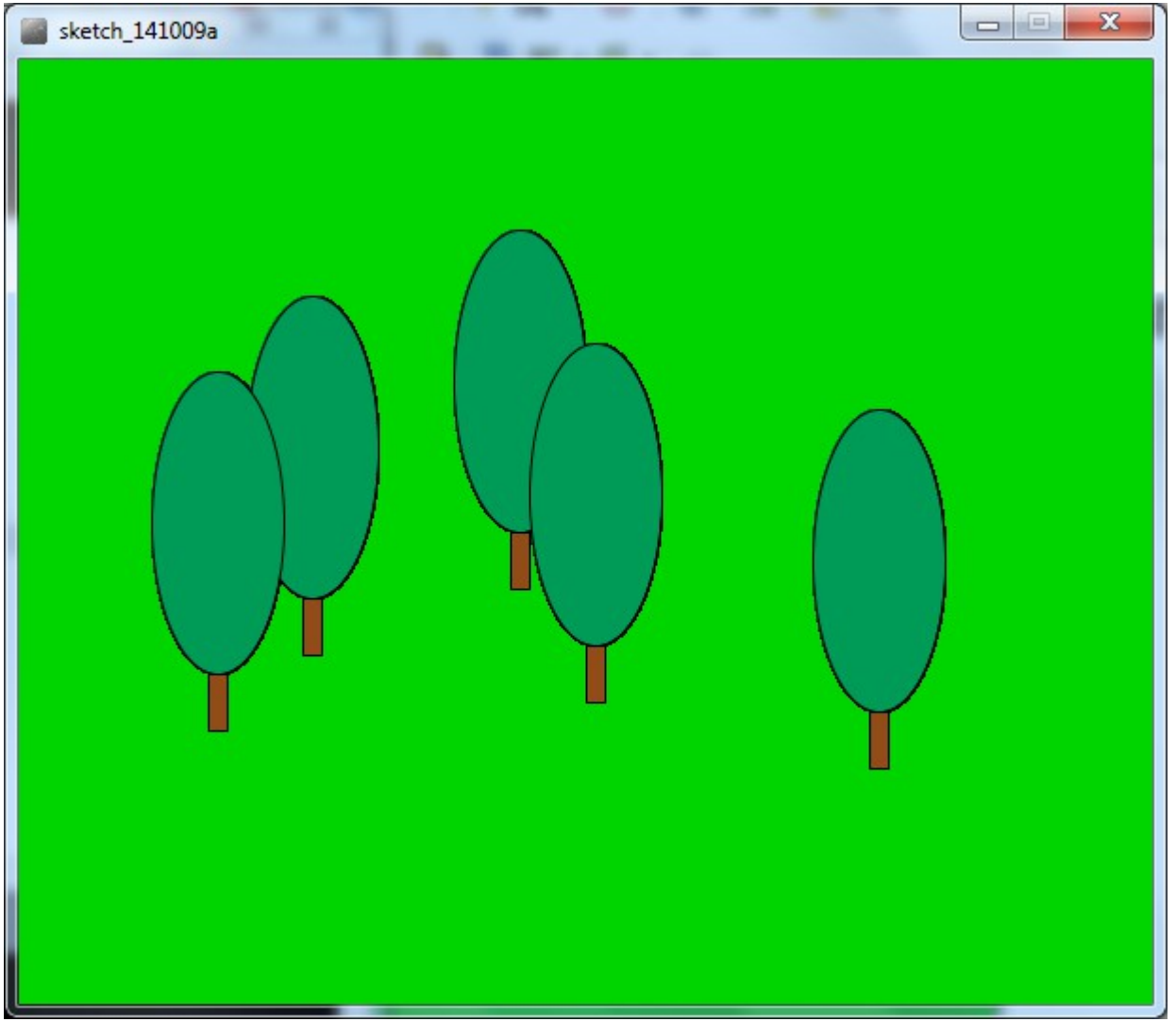
Ein Referenzattribut muss deklariert werden, ein Objekt der Klasse muss erzeugt und diesem Attribut zugewiesen werden. Erst danach können Methoden dieses Objekts aufgerufen werden.

Damit sieht unser Hauptprogramm wie folgt aus (aktuell mit nur einem Rechteck):

```
RECHTECK kasten1;              // Deklaration des ersten Rechtecks

void setup()
{
    size(600,600);             // Fenstergroesse
    background(#000000);       // Hintergrundfarbe setzen, hier hexadezimal

    kasten1 = new RECHTECK();  // Erzeugen des Objekts kasten1 – der
                               // Konstruktor initialisiert die Attribute
}

void draw()
{
     kasten1.Zeichne();        // Aufruf der Methode Zeichne des Objekts
}
```

Arbeitsauftrag: Sorgen Sie dafür, dass im Programm vier Objekte mit den Namen kasten1 bis kasten4 existieren. Lassen Sie Ihr Programm ablaufen. Machen Sie sich Gedanken darüber, warum Sie nur ein Rechteck sehen.

Objektorientierte Modellierung – Objektorientierte Programmierung



Modellierung - Konzepte

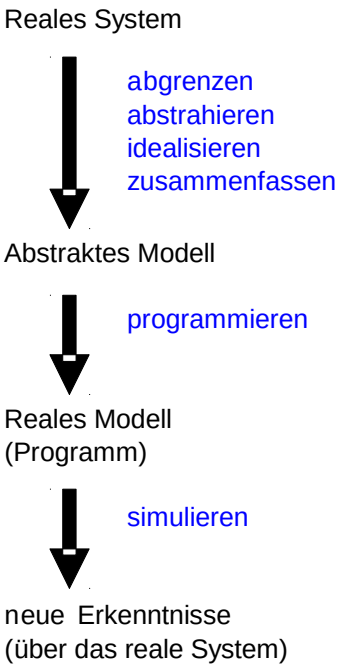
Sich ein „Modell bilden“ bedeutet:

einen Ausschnitt aus dem realen System wählen und alle Einflüsse von außen weglassen

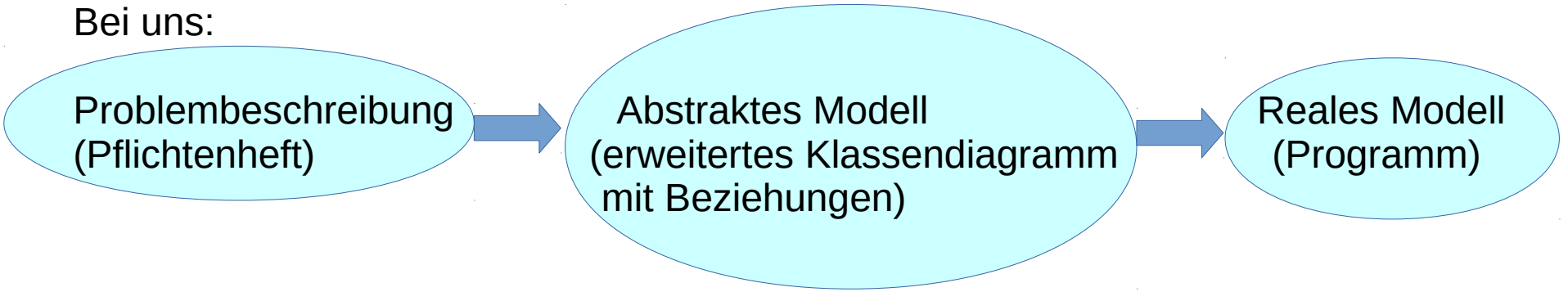
nur die wichtigen Dinge in dem Ausschnitt betrachten

diese wichtigen Dinge so einfach wie möglich und so umfassend wie nötig wiedergeben

das Modell mit einer **normierten** Darstellungsform beschreiben



Bei uns:

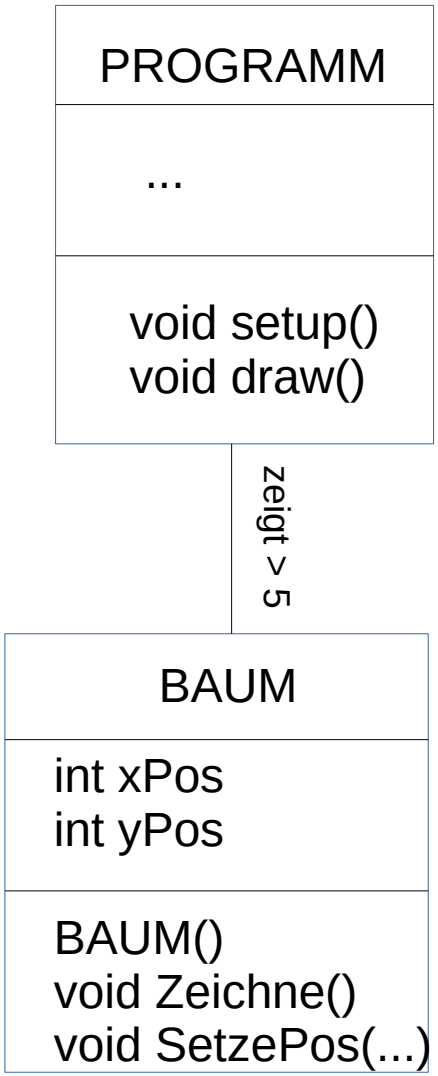


Objektorientierte Modellierung

Unsere Zeichnung enthält fünf Objekte der Klasse BAUM. Sie unterscheiden sich nur in ihrer Position.

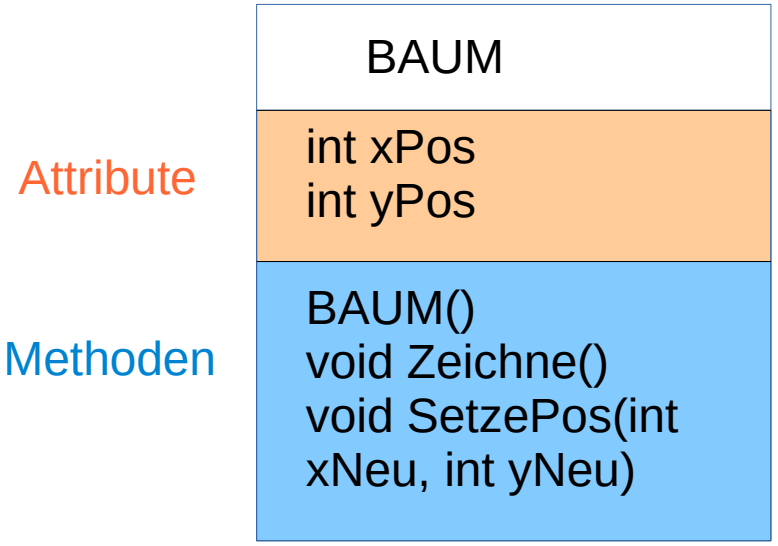
Was muss für diese Objekte möglich sein?

Objektorientierte Modellierung
Wald



erweitertes
Klassendiagramm

Objektorientierte Modellierung
Wald



erweitertes Klassendiagramm

```
public class BAUM
{
    int xPos;
    int yPos;

    BAUM()
    {     ...
    }

    void Zeichne()
    {
          ....
     }

    void SetzePos(int xNeu, intyNeu)
    {
        xPos = xNeu;
        yPos = yNeu;
     }
}
```

Java/Processing-Klasse

Umsetzung der Klasse BAUM in Java/Processing:

```
class BAUM                          // Klassenkopf
{
    int xPos;                       // Attributdeklarationen
    int yPos;

    BAUM()      // Konstruktor BAUM zum Erzeugen eines Objekts
    {
        xPos = 100;           // Setzen der Startwerte der Attribute
        yPos = 100;           // Initialisieren
     }

     void SetzePos( int xNeu, int yNeu)      // Methodenkopf von SetzePos
     {
         xPos = xNeu;         // die Werte der Attribute werden
         yPos = yNeu;         // auf die Parameterwerte gesetzt
      }

      void Zeichne()   // die linke untere Ecke des Baumstamms ist unser Bezugspunkt
      {
          fill(0,150,80);                          // gruen
          ellipse(xPos+5,ypos-110,70,160);
          fill(140,70,20);
          rect(xPos,yPos-30,10,30);
       }
}
```

Einbinden mehrerer Bäume in das Programm - Referenzattribute

```
BAUM baum1;                 // Deklaration eines Referenzattributs

void setup()                // Grundeinstellungen
{
        size(600,600);
        background(0,210,0);
        baum1 = new BAUM();     // Aufruf des Konstruktors zur Erzeugung des Objekts
}

void draw()             // Anweisungen in dieser Methode werden wiederholt ausgeführt
{
          baum1.Zeichne();          // baum1 ist eine Instanz der Klasse, Aufruf der
                                    // Methode Zeichne():

}
```

Ein Referenzattribut muss deklariert werden, ein Objekt der Klasse muss erzeugt und diesem Attribut zugewiesen werden. Erst danach können Methoden dieses Objekts aufgerufen werden.

Arbeitsauftrag:

1. Öffnen Sie ein neues Projekt mit den Namen WALD und schreiben Sie eine passende setup()- und draw()-Methode (noch ohne Referenzattribut).

2. Öffnen Sie in der Processing-Umgebung einen weiteren Tab und geben Sie als Name der Klasse BAUM ein.

3. Schreiben Sie in diesem Tab die Klasse BAUM.

4. Ergänzen Sie im Tab WALD ein Referenzattribut baum1 und erweitern Sie Ihr Programm so, dass baum1 gezeichnet werden kann.

5. Ergänzen Sie weitere Referenzattribute baum2, baum3, baum4, baum5. Achten Sie darauf, die Methode SetzePos sinnvoll einzusetzen, so dass die Bäume an verschiedenen Positionen gezeichnet werden.

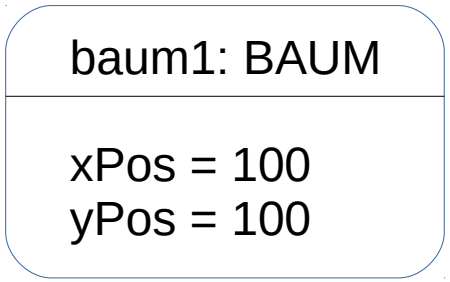
6. Erweitern Sie die Klasse BAUM so, dass sie Bäume verschiedener Farbe zeichnen lassen können. (Tipp: neues Attribut bei BAUM)

Methodenaufrufe – Attributwerte ändern

Für die letzten Aufgaben haben Sie Methodenaufrufe gebraucht, um die Werte der Attribute zu ändern, so dass dann z.B. der Baum baum1 an einer anderen Position gezeichnet werden kann. Damit haben Sie den **Zustand** des Objekts baum1 geändert.

Methodenaufrufe bewirken eine **Zustandsänderung.**

vor dem Aufruf:



Aufruf
baum1.SetzePos(200,150)

nach dem Aufruf: