## Wir lassen zeichnen

In Konstruktionsbüros findest du Plotter wie in Abb. 1. Bei diesen fahren Zeichenstifte computergesteuert mit großer Geschwindigkeit über ein Blatt Papier. Zwischendurch wird der Stift kurz hochgefahren; dann kann der Stift an eine andere Stelle gefahren werden, ohne dass eine Linie dabei gezeichnet wird. Dort wo eine neue Linie beginnen soll, wird der Stift wieder abgesenkt. Manchmal tauscht der Plotter auch seinen Stift aus, um in einer neuen Farbe weiter zu zeichnen.

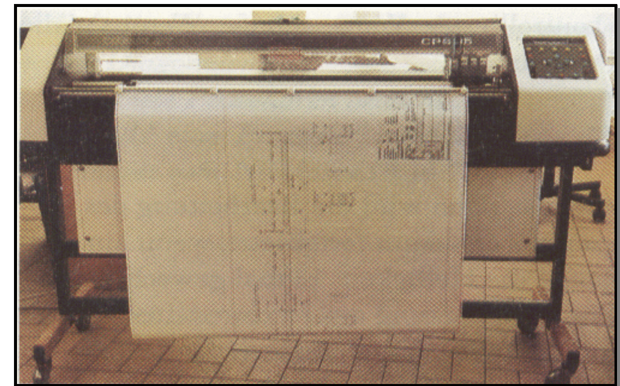


**Abb. 1**: Ein Plotter

Bei vielen Plottern werden übrigens sowohl Stift als auch Papier bewegt: Während der Stift sich nur in x-Richtung bewegt, wird das Blatt Papier über eine Trommel unter dem Stift hinweg in y-Richtung hin- und her bewegt; auf diese Weise können große Blätter mit relativ kleinen Geräten bearbeitet werden.

Wie steuert man nun ein solchen Plotter? Dieser Frage wollen wir in diesem Kapitel nachgehen. Allerdings werden wir keinen richtigen Plotter mit echten Papierbögen benutzen; vielmehr greifen wir auf ein kleines Programm zurück, welches auf eine “virtuelle” Zeichenfläche auf dem Bildschirm malt.

## Ein Igel zeichnet

Im Verzeichnis `source/igel` der CD findest du die Datei `igel.htm`. Wenn du sie doppelt anklickst startet der Browser und du siehst eine Webseite wie in Abb. 2. Allerdings sind sowohl die Zeichenfläche als auch der Eingabebereich für die Kommandos noch leer.

Wir stellen uns nun einen kleinen Igel[1] vor, der auf der Zeichenfläche herumlaufen kann. Dabei folgt er gehorsam den Anweisungen, die in dem Eingabebereich eingetragen werden. Der Igel hat den Namen “i”; und so beginnen alle Befehle mit einem „i“. Der Igel wandert z.B. 30 Schritte vorwärts, wenn man die Anweisung `i.vw(30)` gibt. Die Zahl in den Klammern bezeichnet man als einen **Parameter**. Die Parameterwerte müssen nicht unbedingt ganzzahlig sein. Allerdings ist zu beachten, dass als Dezimalzeichen kein Komma, sondern ein Punkt benutzt wird.

Der Igel ist mit einem Stift ausgerüstet, der gewöhnlich auf die Zeichnefläche gesenkt ist. Wenn der Igel sich nun bewegt, hinterlässt er eine entsprechende Spur. Man kann dem Igel auch mitteilen, dass er den Stift anheben soll; das geschieht mit dem Befehl `i.stifthoch()`. Dann hinterlässt er beim Laufen keine Linie. Soll der Igel den Stift wieder auf die Zeichenfläche

[1]Unser Igel wurde 1968 in Amerika von S. Papert als “turtle” (engl. für Schildkröte) erfunden. In der deutschsprachigen Literatur spricht man allerdings meistens von einem Igel.

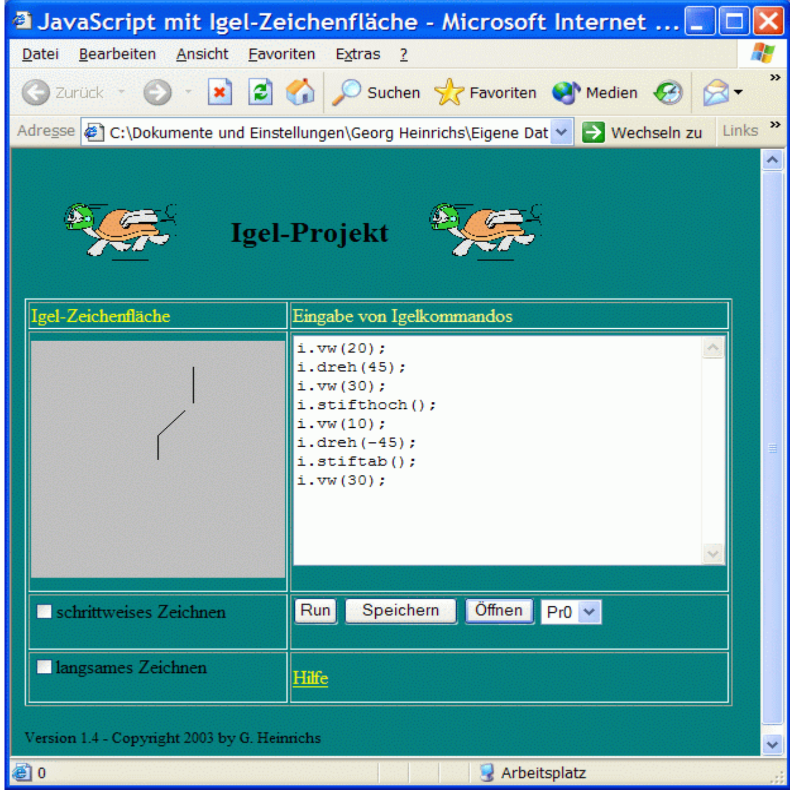


**Abb. 2**: Der Igel folgt gehorsam allen Anweisungen

zeichnen, muss man den Befehl `i.stiftab()` geben. Nach dem Start der Igel-Seite ist der Stift zunächst immer abgesenkt, der Igel blickt in Nordrichtung (nach oben) und befindet sich in der Mitte der Zeichenfläche.

Und nun kommt der große Augenblick: Wir geben unsere erste Igel-Anweisung ein:

```
i.vw(20)
```

Zunächst geschieht nichts; erst wenn wir den "Run"-Knopf betätigen, geht der Igel blitzartig 20 Schritte vorwärts und hinterlässt eine entsprechend lange gerade Linie als Spur.

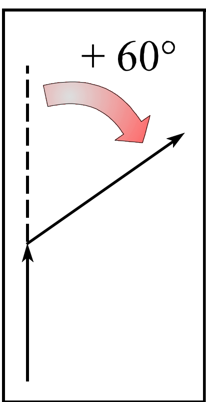


**Abb. 3**: Hier dreht sich der Igel um 60°.

Wir geben in die nächsten beiden Zeilen zwei weitere Befehle ein:

```
i.dreh(90)
i.vw(40)
```

Nach dem Betätigen der Run-Taste sehen wir einen rechten Winkel, dessen Schenkel gleich lang sind. Auf den ersten Blick erscheint das seltsam, denn nach Norden sollte der Igel nur 20 Schritte, nach Osten hingegen 40 Schritte gehen. Das Verhalten des Igels erklärt sich aus folgender Regel:

Wird der Run-Knopf betätigt, so werden ***alle*** Anweisungen im Kommandofenster ***der Reihe nach*** abgearbeitet. Beim ersten Mal (oder nach dem Löschen der Zeichnung mit der Aktualisieren-Taste ) startet der Igel im Zentrum der Zeichenfläche mit der Blickrichtung nach oben (Nordrichtung). Sonst startet der Igel bei der aktuellen Position mit der aktuellen Blickrichtung.

Aktualisieren wir jetzt die Webseite und betätigen erneut den Run-Knopf! Jetzt geht der Igel tatsächlich 20 Schritte nach Norden, dreht sich um 90° nach rechts (im umgangssprachlichen und nicht im mathematischen Sinne) und geht anschließend 40 Schritte nach Osten.

Unsere Folge von drei Igel-Anweisungen stellt bereits ein kleines Programm dar. Unter einem **Programm** versteht man eine Verarbeitungsvorschrift, welche aus einer endlichen Folge von Anweisungen besteht. In der Abb. 2 siehst du ein etwas längeres Programm. Versuche einmal, anhand der Zeichnung die Bedeutung der einzelnen Anweisungen nachzuvollziehen.

In der folgenden Tabelle sind alle Igelbefehle zusammengefasst und kurz erläutert. Beachte, dass unser Igel sehr pingelig ist: So mag er es gar nicht, wenn z.B. die Klammern bei der Stifthoch-Anweisung vergessen oder kleine mit großen Buchstaben (und umgekehrt) vertauscht werden. Außerdem versteht er pro Zeile normalerweise nur einen Befehl. Willst du ihm in einer Zeile mehr als einen Befehl geben, musst du sie durch ein Semikolon trennen.

| **Anweisung** | **Bedeutung** |
|---|---|
| `i.stifthoch()` | Der Stift wird hochgehoben. Wenn der Igel mit i.vw(...) bewegt wird, wird keine Linie gezeichnet. |
| `i.stiftab()` | Der Stift wird auf die Zeichenfläche hinabgeführt. Wenn der Igel mit i.vw(...) bewegt wird, wird eine Linie gezeichnet. |
| `i.vw(strecke)` | Der Stift wird um die angegebene Strecke in der aktuellen Richtung bewegt. Die Richtung beim Start ist nach oben (Norden). Dabei wird gegebenenfalls eine Linie gezeichnet. Ist der Wert der Variablen “strecke” negativ, so geht der Igel rückwärts. |
| `i.dreh(winkel)` | Der Igel wird um den angegebenen Winkel im Uhrzeigersinn gedreht, wenn der Wert der Variablen “winkel” positiv ist, ansonsten entgegengesetzt. |
| `i.setzeFarbe(r,g,b)` | Die Zeichenfarbe wird neu festgelegt; dabei sind r, g und b Zahlen zwischen 0 und 255. Diese Zahlen geben den Rot-, Grün- und Blauanteil an. Genaueres dazu im Intermezzo zur Farbenlehre. |
| `i.warte(t)` | Der Igel wartet t Millisekunden. |

Deine Programme kannst du auch auf der Festplatte sichern; es können bis zu 5 verschiedene Programme abgespeichert werden. Dazu wählt man zunächst die gewünschte Programmnummer mit dem Programm-Optionsfeld Pr0 aus und betätigt anschließend die "Speichern"- oder "Öffnen"-Schaltfläche. Es steht insgesamt nur ein Speicherplatz von maximal 4 kB zur Verfügung. Sollte man einmal mehr Speicherplatz belegen, so lassen sich die Programme erst dann wieder laden, wenn ein oder mehrere Programme gelöscht worden sind. Das Löschen geschieht durch Speichern eines leeren Kommandofensters.

### *Aufgaben*

1. Der Igel soll ein Quadrat mit der Seitenlänge 40 zeichnen.

2. Zeichne wieder ein Quadrat wie in Aufgabe 1. Diesmal soll allerdings das Quadrat mittig auf der Zeichenfläche gezeichnet werden. Benutze `i.stifthoch()`.

3. Zeichne ein gleichseitiges Dreieck mit der Seitenlänge 50.

4. Zeichne das "Haus vom Nikolaus" (vgl. Abb. 4).

5. Zeichne das "Haus vom Nikolaus" farbig.

6. Probiere aus: Welche Bedeutung haben die Kontrollkästchen "schrittweises Zeichnen" und "langsames Zeichnen"?

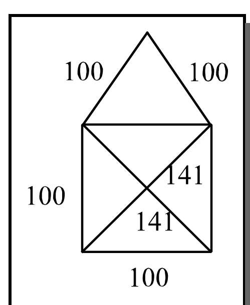



**Abb. 4**: Zu Aufgabe 4

7. Auch beim Fernsehen gibt es ein „Programm". Vergleiche dieses mit unseren Igelprogrammen. Nenne Gemeinsamkeiten und Unterschiede.

## Funktionen

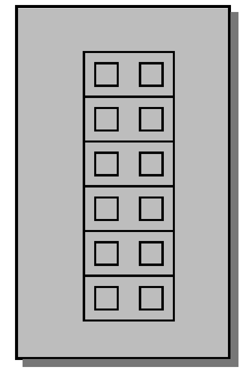


**Abb. 5**: Hochhaus

In vielen Konstruktionszeichnungen gibt es Teile von Zeichnungen, die sich wiederholen, z.B. die Fenster bei einem Hochhaus wie in Abb.5. Sicherlich muss es einen einfacheren Weg geben, als die Folge von Anweisungen für jedes Fenster 12 mal einzugeben (oder über die Zwischenablage entsprechend oft zu kopieren). Wie das geht, wollen wir an einem einfachen Beispiel zeigen.

Drei Quadrate mit der Seitenlänge 20 sollen übereinander gezeichnet werden. Ein einzelnes dieser Quadrate wird durch die Anweisungen

```
i.vw(20); i.dreh(90);
i.vw(20); i.dreh(90);
i.vw(20); i.dreh(90);
i.vw(20); i.dreh(90);
```

gezeichnet. Diese Folge von Anweisungen fassen wir jetzt zu einem sogenannten **Anweisungsblock** zusammen; dies geschieht, indem wir ihn in geschweifte Klammern setzen:

```
{
     i.vw(20); i.dreh(90);
     i.vw(20); i.dreh(90);
     i.vw(20); i.dreh(90);
     i.vw(20); i.dreh(90);
}
```

Zusätzlich geben wir dem Block den Namen `quadrat20`. Dazu schreiben wir vor den Block die Zeile

```
function quadrat20()
```

Dadurch haben wir eine neue Anweisung programmiert. Wenn wir jetzt nämlich die Zeile

```
quadrat20()
```

eintippen, wird ein Quadrat mit der Seitenlänge 20 gezeichnet. Eine derart programmierte Anweisung wird als **Funktion** bezeichnet.

Allgemein unterscheidet man zwischen der Definition und dem Aufruf einer Funktion. Die Definition einer Funktion wird eingeleitet durch das Schlüsselwort **`function`**. Es folgt der **Funktionsname**; dieser muss mit den beiden runden Klammern abgeschlossen werden, wie wir sie schon von den Igel-Anweisungen `i.stifthoch()` und `i.stiftab()` kennen. Das Schlüsselwort `function` und der Funktionsname bilden den **Funktionskopf**. Dahinter steht der sogenannte **Funktionsrumpf**, der aus einem Anweisungsblock besteht (Abb. 6).

| function quadrat20() | Funktionskopf |
|---|---|
| {<br>    i.vw(20); i.dreh(90);<br>    i.vw(20); i.dreh(90);<br>    i.vw(20); i.dreh(90);<br>    i.vw(20); i.dreh(90);<br>} | Funktionsrumpf |

**Abb. 6**: Aufbau einer Funktion

Eine bereits definierte Funktion kann man nun **aufrufen**, indem man einfach ihren Namen eingibt. Dann werden der Reihe nach alle Anweisungen ausgeführt, die zum Funktionsrumpf gehören. Um drei Quadrate übereinander zu zeichnen, brauchen wir also nur noch die Anweisungen

```
quadrat20(); i.vw(20);
quadrat20(); i.vw(20);
quadrat20();
```

eingeben.

Die selbst definierten Funktionen können also genauso eingesetzt werden wie die bereits bekannten Anweisungen. Insbesondere können sie selbst wieder Bestandteile des Funktionsrumpfes einer anderen Funktion werden. So könnten unsere drei Quadrate auch durch die Funktion

```
function dreiquadrate()
{
     quadrat20(); i.vw(20);
     quadrat20(); i.vw(20);
     quadrat20();
}
```

gezeichnet werden. Beachte dabei aber, dass deine selbstdefinierten Funktionen nur so lange zur Verfügung stehen, wie sie im Kommandofenster stehen. Zum Speichern kannst du die Speichern-Schaltfläche oder folgenden Trick benutzen:

Den Inhalt des Kommandobereichs kannst du folgendermaßen auf Diskette oder Festplatte abspeichern: markiere den gewünschten Bereich, kopiere ihn in die Zwischenablage. Öffne nun einen Texteditor (z.B. Notepad) und füge den Inhalt der Zwischenablage ein. Speichere den Text nun an der gewünschten Stelle ab; benutze dabei die für JavaScript-Programme übliche Extension “js”. Auf umgekehrten Weg kannst du die Programme auch wieder in den Kommandobereich laden.

***Aufgaben***

1. Gib die Funktion `quadrat20()` im Kommandobereich ein und teste sie aus.

2. Jetzt sollen drei Quadrate nebeneinander gezeichnet werden. Schreibe dazu auch eine Funktion und teste sie aus.

3. In HTML gibt es auch Kopf und Rumpf. Durch welche Tags werden sie gekennzeichnet.

4. Schreibe eine Funktion `dreieck20()`, welche ein gleichseitiges Dreieck mit der Seitenlänge 20 zeichnet.

5. Schreibe Funktionen `haus()` und `doppelhaus()` zu der Abbildung 7. Benutze die Funktionen aus den Aufgaben 1 und 4.

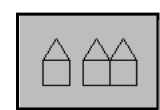


Abb. 7

```
function wasZeichneIch1()
{
    quadrat50();
    i.vw(50);
    dreieck50();
    i.vw(-50);
}

function quadrat50()
{
    i.vw(50); i.dreh(90);
    i.vw(50); i.dreh(90);
    i.vw(50); i.dreh(90);
    i.vw(50); i.dreh(90);
}

function dreieck50()
{
    i.dreh(30);
    i.vw(50); i.dreh(120);
    i.vw(50); i.dreh(120);
    i.vw(50); i.dreh(120);
    i.dreh(-30);
}
```

```
function wasZeichneIch2()
{
i.vw(50); i.dreh(30);
i.vw(50); i.dreh(120);
i.vw(50); i.dreh(120);
i.vw(50); i.vw(-50);
i.dreh(-90); i.vw(50);
i.dreh(90); i.vw(50);
}
```

**Abb. 8**: Verschiedene Programmierstile

## Übersichtliches Programmieren

Schau dir die beiden Programme in Abb. 8 an! Kannst du erkennen, was hier gezeichnet wird? Offensichtlich fällt die Antwort bei dem linken Programm viel einfacher, weil es viel übersichtlicher gestaltet ist. Gerade bei umfangreicheren Programmen ist es wichtig, für Übersichtlichkeit zu sorgen. So kann man beim Programmieren Fehler vermeiden und später leichter Änderungen oder Ergänzungen vornehmen.

Beim Programmieren solltest du deswegen beachten:

**Zerlege dein Problem** mithilfe von Funktionen in einzelne Teilprobleme. Diese Art der Programmierung nennt man **modulare Programmierung**.

Verwende **sinnvolle/sinngebende Funktionsnamen**. Scheue dabei nicht, längere Namen zu benutzen, auch wenn sie etwas mehr Arbeit bei der Eingabe bedeuten.

Achte darauf, dass am Ende einer Funktion der Igel wieder an derselben Stelle steht wie am Anfang; er sollte auch wieder die ursprüngliche Richtung besitzen. So kann man schwer kontrollierbare **Seiteneffekte verhindern**.

Füge **Kommentare** in dein Programm ein. Diese werden mit // eingeleitet. Alles, was in einer Zeile hinter dem Doppelschrägstrich steht, wird nicht als Anweisung angesehen.

Wie im linken Teil der Abb. 8 solltest du die einzelnen **Anweisungen eines Blockes einrücken**.

Bei der Wahl der Funktionsnamen gibt es allerdings einige Einschränken:

1. Ein Funktionsname darf keine Sonderzeichen (Ausnahme: der Unterstrich „_"), keine Umlaute sowie kein „ß" enthalten.

2. Ein Funktionsname darf nicht mit einem reservierten Schlüsselwort übereinstimmen. Eine Liste dieser Wörter findest du im Anhang.

***Aufgaben***

1. Warum ist es wichtig, beim Programmieren für Übersicht zu sorgen?

2. Gib die Funktion `quadrat50()` ein. Durch die Anweisungsfolge

   ```
   quadrat50(); i.vw(50); quadrat50();
   ```

   werden zwei Quadrate übereinander gezeichnet. Warum erhält man ein anderes Ergebnis, wenn man in der Quadratfunktion die letzte Zeile weglässt?

3. Zeichne das Ergebnis der Funktion `wasZeichneIch2()` aus der Abb. 8.

4. Zeichne ein Rad mit 3 Flügeln. Programmiere modular!

5. Schlage im Lexikon nach oder suche im Internet:
   a) Welche Bedeutung hat das Wort *Modul*?
   b) Im Zusammenhang mit dem modularen Programmieren spricht oft von der Bottom-Up- und von der Top-Down-Methode. Was versteht man darunter?

**Funktionen mit Parametern**

Unsere bisher definierten Funktionen haben einen Nachteil: sie sind nur eingeschränkt einsetzbar: Wenn z. B. ein Quadrat mit der Seitenlänge 30 gezeichnet werden soll, hilft uns die Funktion `quadrat20()` aus dem vorletzten Abschnitt recht wenig. Wünschenswert wäre eine Flexibilität, wie wir sie schon bei der Igel-Anweisung `i.vw(x)` kennengelernt haben. Wie viele Schritte der Igel vorwärts geht, hängt davon ab, welchen Wert wir für x einsetzen.

In der Tat lassen sich derartige Funktionen mit Parameter leicht programmieren. Die folgende Funktion zeichnet z.B. Quadrate beliebiger Seitenlänge:

| **Definition** | **Aufruf** |
|---|---|
| `function quadrat(s)`<br>`{`<br>`    i.vw(s); i.dreh(90);`<br>`    i.vw(s); i.dreh(90);`<br>`    i.vw(s); i.dreh(90);`<br>`    i.vw(s); i.dreh(90);`<br>`}` | `quadrat(30)`<br>Der Wert 30 wird an die Funktion übergeben; überall im Funktionsrumpf wird s durch 30 ersetzt. Dann werden die Anweisungen ausgeführt. |

Im Unterschied zu den bisher betrachteten Funktionen ist die Klammer hinter dem Funktionsnamen nicht leer; vielmehr steht dort eine Variable. Das ist ein Platzhalter für einen Parameterwert. Wenn nun der Funktionsaufruf `quadrat(30)` erfolgt, wird der Parameterwert 30 an die Funktion `quadrat(s)` **übergeben**. Dabei wird im Funktionsrumpf die Variable s an jeder Stelle durch diesen Wert 30 ersetzt. Anschließend werden die Anweisungen des Funktionsrumpfes ausgeführt.

Die Variablennamen können auch aus mehreren Buchstaben bestehen; für sie gelten dieselben Regeln wie für die Namen von Funktionen:

**Namen von Variablen und Funktionen** beginnen mit Buchstaben (außer Umlaut oder ß!) oder mit einem Unterstrich. Als weitere Zeichen können außerdem auch Ziffern benutzt werden. Die Namen dürfen nicht mit einem Schlüsselwort (s. Anhang) übereinstimmen.

Eine sinnvolle Wahl der Variablennamen kann das Programm übersichtlicher machen. So ist bei dem Funktionskopf

```
function rechteck(a, b)
```

nicht klar, welche Seite im Rechteck a und welche b sein soll. Viel deutlicher ist, wenn die die Variablennamen `hoehe` und `breite` benutzt werden. Damit sieht die Funktion so aus:

```
function rechteck(hoehe, breite)
{
     i.vw(hoehe); i.dreh(90);
     i.vw(breite); i.dreh(90);
     i.vw(hoehe); i.dreh(90);
     i.vw(breite); i.dreh(90);
}
```

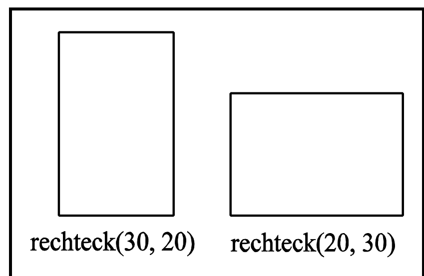



**Abb. 9**: Die Reihenfolge der Parameter ist wichtig.

Dieses Beispiel zeigt auch: Eine Funktion kann auch zwei oder mehr Parameter besitzen. In solchen Fällen müssen die Parameter natürlich in der richtigen Reihenfolge eingegeben werden (vgl. Abb. 9).

***Aufgaben***

1. Schreibe eine Funktion `quadrat(x)`, welche ein Quadrat zeichnet. Teste sie aus. Benutze dabei auch negative Parameterwerte. Was fällt auf?

2. Schreibe die Funktionen `dreieck(x)` für ein gleichseitiges Dreieck mit der Seitenlänge x.

3. Schreibe eine Funktion `haus(x)` und eine Funktion `doppelhaus(x)`, welche Häuser wie in der Abb. 10 mit der Seitenlänge x zeichnet.

**Abb. 10**

4. Programmiere den Tannenbaum aus Abb. 11 zunächst mit den angegebenen Maßen. Benutze die Dreiecksfunktion aus Aufgabe 2.
   Schreibe dann eine Funktionen `tannenbaum(z1, z2, z3)`, bei der die Größe der Dreiecke variabel sind.

5. Wie lässt sich mithilfe der Rechteckfunktion eine Quadratfunktion programmieren?

15
23
30

**Abb. 11**

**Variablen**

Für die Mauer in Abb. 12 benötigt man eine Funktion `stein(hoehe)`. Diese soll Rechtecke zeichnen, deren Breite doppelt so groß wie die Höhe ist. Am einfachsten ist es, hierzu die bereits bekannte Funktion `rechteck(hoehe, breite)` zu benutzen. Allerdings muss zuerst noch die Breite ausgerechnet werden. Das Ergebnis muss das Programm sich merken und dann als (zweiten) Parameter an die Funktion `rechteck` übergeben. Dies geschieht folgendermaßen:

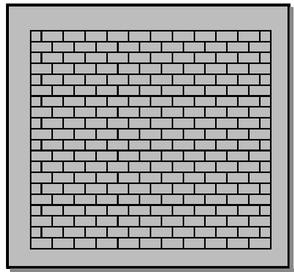


**Abb. 12**: Eine Mauer

```
var b;
b = hoehe * 2;
```

In der ersten Zeile teilt das Schlüsselwort **`var`** dem Computer mit, dass eine Variable mit dem Namen `b` deklariert wird. In der zweiten Zeile wird der Wert des Parameters `hoehe` mit 2 multipliziert und die Variable `b` erhält diesen Wert.

Das klingt ziemlich abstrakt. Mit einem einfachen Modell – dem so genannten **Schubladenmodell** – können wir es veranschaulichen. Wir stellen uns vor, der Computer besitzt einen großen Schrank mit einer Vielzahl von Schubladen (vgl. Abb. 13). In diese Schubladen steckt er alles, was er sich merken soll: Texte, Zahlen oder auch Bilder und Musikstücke. Und damit er diese auch wiederfinden kann, besitzen alle Schubladen, in denen schon etwas gespeichert ist, ein Etikett mit einer eindeutigen Kennzeichnung.

Wenn die Anweisung `var b` ausgeführt wird, beschriftet der Computer ein Etikett mit dem **Variablennamen** `b` und klebt es auf eine noch freie Schublade. Das ist die **Variablendeklaration**. Durch die Anweisung `b = 47` wird der Computer nun dazu veranlasst, die Zahl 47 auf einen Zettel zu schreiben und ihn in diese Schublade zu stecken (Abb. 13). Dies bezeichnet man als **Speicher- oder Schreibvorgang**. Was abgespeichert wird, steht rechts vom Gleichheitszeichen; wo es abgespeichert wird, steht links vom Gleichheitszeichen. Die Anweisung `47 = b` ist demnach sinnlos und wird vom Rechner mit einer Fehlermeldung quittiert.

Allerdings können Variablennamen auch rechts vom Gleichheitszeichen auftauchen. Das ist z.B. in der Anweisung `b = hoehe * 2` der Fall. In diesem Fall erfolgt ein **Lesevorgang**: Die Schublade mit dem Etikett `hoehe` wird geöffnet und der Inhalt gelesen; dabei wird der Zettel mit dem Wert nicht aus der Schublade entfernt. Nehmen wir an, in der Schublade ist der Wert 23 gespeichert. Dann wird diese Zahl mit 2 multipliziert und das Ergebnis – also 46 – anschließend in unsere Schublade mit dem Etikett `b` gelegt.

Die Ausdrücke auf der rechten Seite des Gleichheitszeichen können auch komplizierter sein, z. B. Terme mit verschiedenen Zahlen, Variablen, Rechenzeichen und auch Klammern. Dabei gelten die üblichen Rechenregeln.

Dass man als Anweisung zum Speichern gerade ein Gleichheitszeichen benutzt, kann manchmal etwas verwirrend sein. Die Anweisung

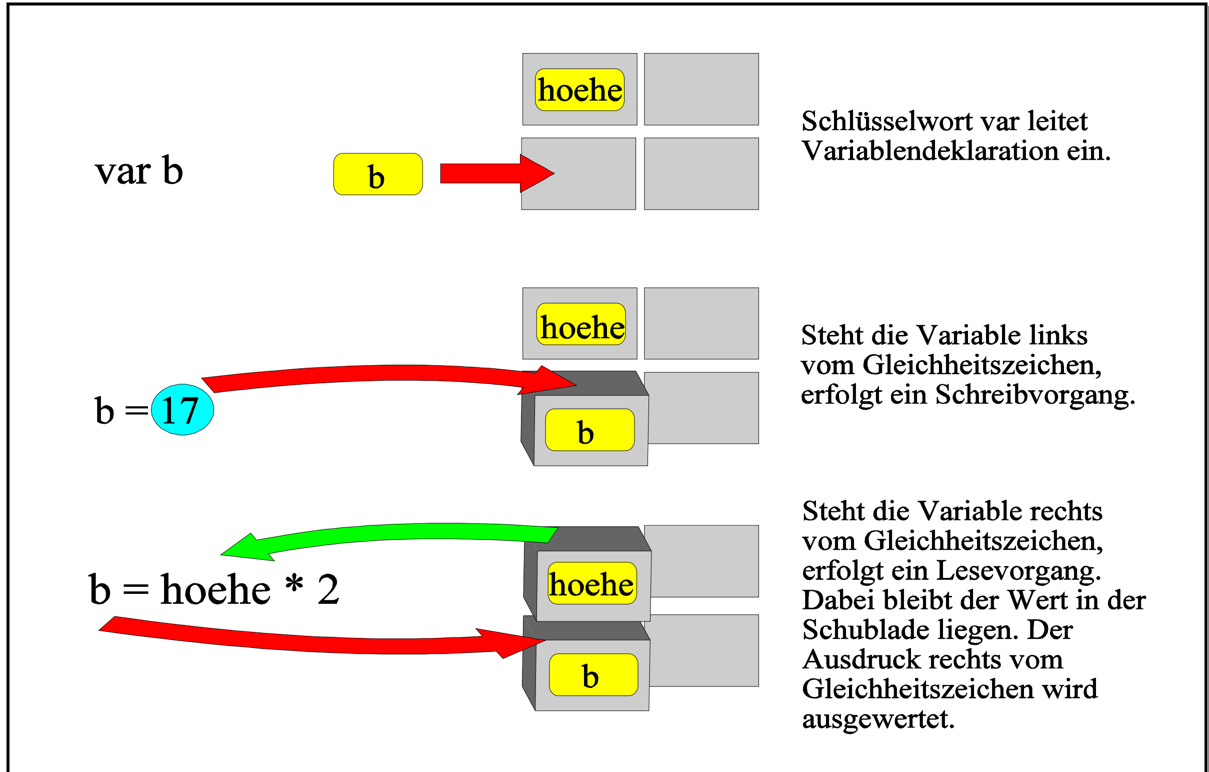



**Abb. 13**: Deklaration, Schreib- und Lesevorgang im Schubladenmodell

```
x = x + 1
```

ist keine Gleichung, wie sie in der Mathematik zu finden ist; dort hätte sie keine Lösung. Hier bedeutet sie vielmehr: Lies den Wert aus der Schublade mit dem Etikett `x`, addiere 1 zu diesem Wert und speichere das Ergebnis wieder in der Schublade `x` ab; dabei wird der alte Zettel weggeworfen. Kurz gesagt wird der Wert in dieser Schublade um 1 erhöht.

Die Deklaration einer Variablen kann auch direkt mit einer Wertzuweisung verbunden werden. Durch `var x1 = 37.6` wird eine Schublade mit dem Etikett `x1` versehen und mit dem Wert 37,6 gefüllt.

### *Aufgaben*

1. Beschreibe im Schubladenmodell, welche Aktionen durch die Zeilen

   ```
   var x; var y;
   x = 70; y = 100;
   x = y - x
   ```

   ausgeführt werden.

2. Tippe die Zeilen aus Aufg. 1 im Kommandobereich ein. Füge die Anweisung `i.vw(x)` an, betätige den Run-Knopf. Entspricht das Ergebnis deinen Erwartungen?

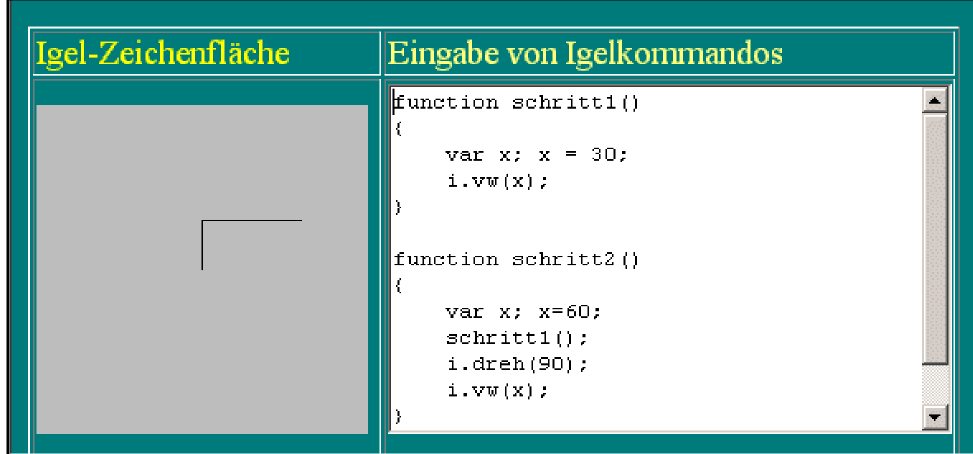



**Abb. 14**: So verhält sich der Igel beim Ausführen dieses Programms.

## Globale und lokale Variable

Wird die Funktion `schritt2()` aus Abb. 14 aufgerufen, so wird ein Winkel mit den Schenkellängen 30 und 60 gezeichnet. Mit unserem bisherigen Schubladenmodell können wir dies nicht erklären: Nachdem zu Beginn der Funktion `schritt2()` eine Schublade mit dem Etikett `x` versehen worden ist und den Wert 60 erhalten hat, wird durch die Funktion `schritt1()` erneut angeordnet, eine Schublade mit demselben Etikett zu versehen. Das sollte eigentlich schon zu einer Fehlermeldung führen. In Wirklichkeit wird das Programm ohne Probleme ausgeführt.

Es kommt noch schlimmer: Nach der zweiten Deklarierung erhält unsere `x`-Schublade den Wert 30; anschließend geht der Igel 30 Schritte vorwärts. Danach ist die Funktion `schritt1()` vollständig abgearbeitet. Jetzt erfolgt die Drehung um 90° und der Igel sollte x Schritte zurücklegen; da in der Schublade der Wert 30 liegt, sollten dies auch 30 Schritte sein. In Wirklichkeit sind es 60!

Unser Schubladenmodell muss also verbessert werden. Die entscheidende Idee ist: Es gibt nicht nur einen einzigen Schubladenschrank, sondern viele. Genauer gesagt besitzt jede Funktion ihren eigenen privaten Schrank. Auf diese Schubladen kann nur die jeweilige Funktion, nicht aber eine andere zugreifen. In der Abb. 15 kann z.B. auf die Variablen `x1`, `y1` und `z` nur die Funktion `test1()` zugreifen, auf die Variablen `x2`, `y2` nur die Funktion `test2()`. Beiden Funktionen steht auch eine Variable mit dem Namen `a` zur Verfügung, aber trotz des gleichen Namens sind diese Schubladen völlig unabhängig voneinander. Die Variablen aus diesen privaten Schubladenschränken heißen **lokale Variable**. Lokale Variable werden innerhalb ihrer Funktion deklariert, in der Regel zu Beginn des Funktionsrumpfes.

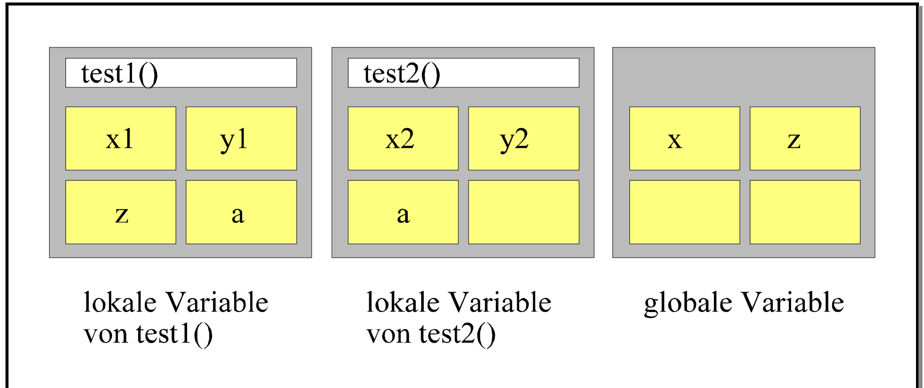



**Abb. 15**: Jede Funktion hat ihren eigenen Variablen-Schrank.

Daneben gibt es auch **globale Variable**. Man kann sich vorstellen, dass diese zu einem weiteren Schrank gehören, der nicht einer speziellen Funktion gehört (Abb. 15). Auf diese globalen Variablen kann im Prinzip von jeder Funktion aus zugegriffen werden, ebenso auch von einzelnen Anweisungen aus, die nicht zu einem Funktionsblock gehören. Globale Variable werden außerhalb von Funktionen deklariert.

Taucht innerhalb einer Funktion ein Variablenname auf, so sucht die Funktion immer zunächst in ihrem privaten Schrank nach einer entsprechenden Schublade. Erst wenn sie dort keine findet, sucht sie bei den globalen Variablen. Deswegen kann in der Abb. 2 zwar die Funktion `test2()` auf die globale Variable `z` zurückgreifen, die Funktion `test1()` aber nicht.

Warum hat man überhaupt lokale Variable eingeführt? Reichen die globalen Variablen nicht aus? In der Tat ließen sich alle Programme so schreiben. Allerdings müsste man immer aufpassen, ob nicht unbeabsichtigt eine Funktion die Zwischenergebnisse einer anderen überschreibt. Durch das Konzept der lokalen Variable kapseln sich die Funktionen nach außen ab; so sind sie vor unbeabsichtigter Manipulation von außen geschützt.

### *Aufgaben*

1. Erkläre das Ergebnis von Abb. 14 mit dem neuen Schubladenmodell.

2. Zu welchem Bild führt das rechts stehende Programm. Erkläre!

3. Welchen Vorteil haben lokale Variable?

```
var c; c = 20;

function schritt1()
{
    var c; c = 40;
    i.vw(c);
}

schritt1();
i.dreh(90);
i.vw(c);
```

**Abb. 16**: Zu Aufgabe 2

## Zählschleifen

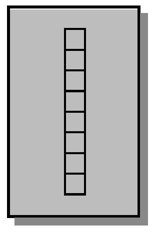
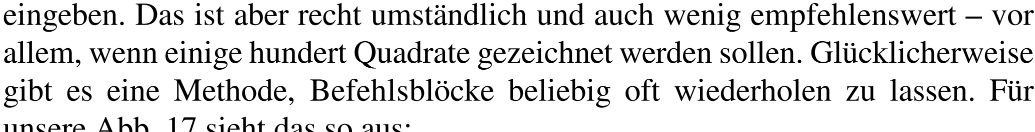


**Abb. 17**

Um einen Stapel von Quadraten wie in Abb. 17 zeichnen zu lassen, kann man natürlich 8 mal hintereinander die Befehle

```
quadrat(10); i.vw(10);
```



eingeben. Das ist aber recht umständlich und auch wenig empfehlenswert – vor allem, wenn einige hundert Quadrate gezeichnet werden sollen. Glücklicherweise gibt es eine Methode, Befehlsblöcke beliebig oft wiederholen zu lassen. Für unsere Abb. 17 sieht das so aus:

```
for (var k = 1; k <= 8; k=k+1)
{
     quadrat(10);
     i.vw(10);
}
```

In der ersten Zeile wird zunächst eine **Zählvariable** k deklariert und mit dem Startwert 1 belegt. Dann wird der folgende Anweisungsblock ein erstes Mal abgearbeitet. Anschließend wird die Zählvariable gemäß der **Schrittanweisung** `k=k+1` um 1 erhöht. Der Anweisungsblock wird ein weiteres Mal abgearbeitet, die Zählvariable erhöht usw.. Dies geschieht solange wie die Zählvariable k kleiner oder gleich 8 ist. Das wird durch die **Durchlaufbedingung** `k<=8` festgelegt.

Eine solche Konstruktion nennt man eine **Zählschleife**. Die erste Zeile bestimmt, wie oft die Schleife durchlaufen wird. Sie wird **Schleifenkopf** genannt. Der Anweisungsblock, welcher wiederholt wird, heißt Schleifenkörper oder **Schleifenrumpf**. Der Startwert bei einer Schleife muss nicht unbedingt 1 sein, auch kann die Zählvariable auf andere Weise verändert werden. So kann durch die Anweisung `k=k-1` auch rückwärts gezählt werden. Einige typische Schrittanweisungen sind in der folgenden Tabelle zusammengefasst.

| **Schrittanweisung** | **Bedeutung** |
|---|---|
| k = k + 1 | k wird um 1 erhöht |
| k = k - 1 | k wird um 1 erniedrigt |
| k = k - 2 | k wird um 2 erniedrigt |
| k++ (Kurzschreibweise) | k wird um 1 erhöht |
| k-- (Kurzschreibweise) | k wird um 1 erniedrigt |

Die allgemeinen Regeln für eine Zählschleife lauten:

```
for (var zv=Anfangswert; Durchlaufbedingung; Schrittanweisung)
{
          Anweisungen
}
```

Nach jedem Schleifendurchlauf, d.h. nach jeder Durchführung des Schleifenrumpfes, wird die Änderung der Zählvariablen `zv` gemäß der Schrittanweisung vorgenommen; dann wird überprüft, ob die Durchlaufbedingung noch erfüllt ist. Wenn sie erfüllt ist, wird ein weiterer Schleifendurchlauf vorgenommen; andernfalls werden die Anweisungen ausgeführt, die hinter dem Schleifenrumpf stehen.

An der Abb. 18 können wir uns die Schleifenkonstruktion noch einmal verdeutlichen: Nachdem der Startwert der Zählvariable auf 10 gesetzt wurde, wird zunächst überprüft, ob die Durchführbedingung erfüllt ist. Ist dies der Fall, wird die Schleife ein erstes Mal durchlaufen. Am Ende der Schleife wird die Zählvariable verändert, in unserem Fall um 1 erniedrigt. Nun erfolgt wieder eine Überprüfung der Durchlaufbedingung. Unsere Lok reist so lange auf der Schleifenbahn, bis diese Bedingung irgendwann nicht mehr erfüllt ist. Erst dann wird die Weiche umgestellt und unsere Lok kann sich auf ihrem Weg weiteren Anweisungen widmen.

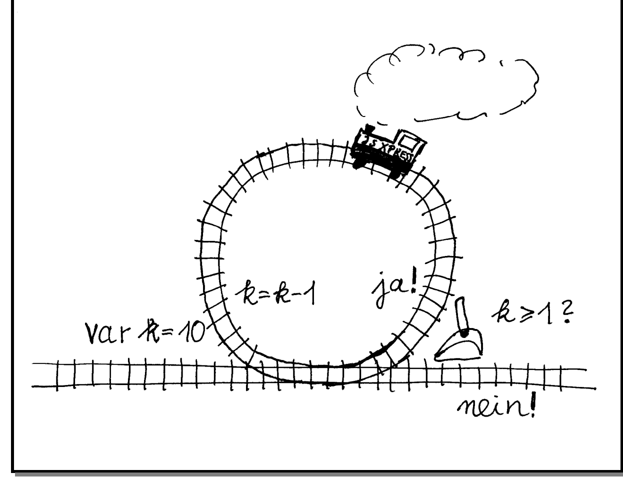


**Abb 18**: Hier wird abwärts gezählt.

Je nach Formulierung der Durchlaufbedingung kann es passieren, dass die Schleife gar nicht durchlaufen oder im Gegenteil niemals mehr verlassen wird. Im letzten Fall spricht man von einer Endlosschleife. Hängt das Programm in einer solchen Schleife fest, kann man nur noch mit der Aktualisieren-Schaltfläche oder der Tastenkombination Strg-Alt-Entf abbrechen. Es lohnt sich also, sorgfältig zu überprüfen, ob die Durchlaufbedingung irgendwann nicht mehr erfüllt ist. Genauso wichtig ist es, die Zählvariable grundsätzlich als lokale Variable zu deklarieren; andernfalls könnte sie vielleicht durch eine andere Funktion unkontrolliert so verändert werden, dass wieder eine Endlosschleife entsteht.

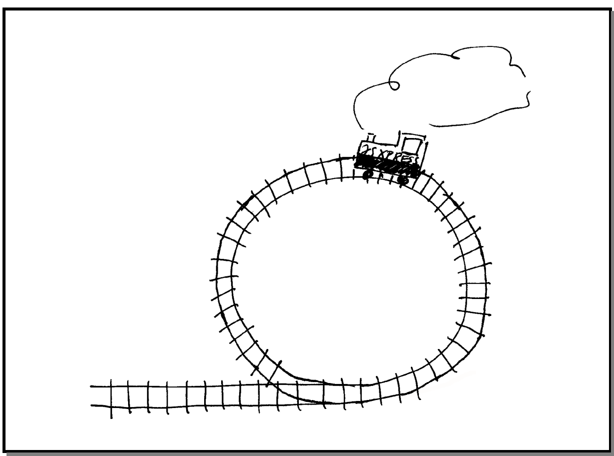


**Abb. 19**: In einer Endlosschleife

***Aufgaben***

1. Eine Reihe von n einfachen Häusern, welche aus einem Quadrat und einem Dreieck der Seitenlänge x bestehen soll gezeichnet werden. Schreibe die Funktion `hausreihe(n, x)` und teste sie aus.

2. Schreibe eine Funktion `quadrat1(x)`, welche ein Quadrat mit der Seitenlänge x zeichnet. Diesmal soll eine Schleife benutzt werden.

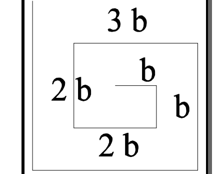


**Abb. 20**

3. Durch die Funktion `turm(anzahl, breite)` soll eine bestimmte Anzahl von Quadraten einer bestimmten Breite übereinander gezeichnet werden.

4. Schreibe die Funktion `n_eck(n, x)`, welche ein gleichseitiges Vieleck mit n Ecken und der Seitenlänge x zeichnet.

5. Schreibe eine Funktion, welche "Spiralen" wie in Abb. 20 erzeugt.

6. Schreibe eine Funktion `tuermchen(n)`, welche eine Zeichnung wie in Abb. 21 erzeugt.

30
40
50

**Abb. 21**

7. Schreibe eine Funktion für das Netz einer Pyramide mit quadratischer Grundfläche. Die Seitenflächen sollen gleichseitige Dreiecke sein. Zeichne erst ein solches Netz in dein Heft und überlege dann, aus welchen elementaren Figuren es besteht. Wie viele Parameter muss die Funktion besitzen?

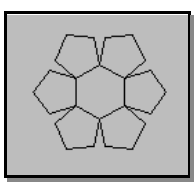


**Abb. 22**

8. Schreibe eine Funktion für das Netz eines Quaders.

9. Schreibe eine Funktion für das Schrägbild eines Quaders.

10. Ein Supereck besteht aus einem gleichmäßigen n-Eck, an dessen Seiten jeweils ein m-Eck angesetzt ist. In Abb. 22 ist ein solches Supereck mit n = 6 und m = 5 zu sehen. Schreibe eine Funktion, die derartige Superecks zeichnen kann. Teste sie für verschiedene Werte von n und m aus.

11. Schreibe eine Funktion `fluegelrad(x, n)`, welche eine Figur wie in Abb. 23 zeichnen soll. Dabei soll n die Anzahl der Flügel und x die (halbe) Länge des Flügels angeben. Überlege zuerst, wie sich die Figur in einfache Bestandteile zerlegen lässt.

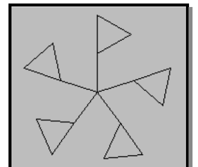


**Abb. 23**

12. Eine Mauer wie in Abb. 12 soll gezeichnet werden. Schreibe eine entsprechende Funktion `mauer(s_hoehe, s_anzahl, z_anzahl)`. Dabei gibt `s_hoehe` die Höhe eines Steins, `s_anzahl` die Anzahl der Steine in einer Mauerzeile und `z_anzahl` die Anzahl der Zeilen an. Beachte dabei, dass es zwei unterschiedliche Arten von Mauerzeilen gibt.

## Animationen

Unter einer **Animation** versteht man eine bewegte Grafik. Häufig bewegt sich dabei nur ein Teil der Objekte vor einem ansonsten unbeweglichen Hintergrund. Um zu erklären, wie eine solche Bewegung programmiert werden kann, wollen wir ein einfaches Beispiel betrachten: Der LKW in Abb. 24 soll von links nach rechts an den Häusern vorbeifahren.

**Abb. 24**: Animation

Als Funktionen brauchen wir zunächst

```
hausreihe_zeichnen()        //stellt Hausreihe dar
auto_zeichnen()             //stellt LKW dar
```

Mit diesen Funktionen werden nun zu Beginn die Hausreihe und das Auto in seiner Startposition gezeichnet.

Wie lässt sich nun das Auto bewegen? Die Idee ist ganz einfach: Wir löschen zuerst das eben gezeichnete Auto, gehen mit dem Igel etwas weiter nach rechts und zeichnen es erneut. Kurze Zeit später löschen wir dieses Auto wieder und zeichnen es etwas weiter nach rechts versetzt noch einmal; das wiederholen wir immer wieder. Sind die Zeitabstände nun klein genug, so nimmt man dies als mehr oder weniger flüssige Bewegung wahr.

Das Schema in Abb. 25 stellt diese Vorgehensweise noch einmal anschaulich dar. Wichtig ist dabei, dass zwischen dem Löschen des Autos und dem Neuzeichnen möglichst wenig Zeit vergeht. Wie gut dies gelingt, hängt von der Schnelligkeit des Rechners ab. Wesentlich länger muss nun das gezeichnete Auto dem Auge des Betrachters präsentiert werden, bis auch dieses wieder gelöscht wird. Allerdings sollte auch diese Wartezeit zwischen dem Zeichnen und Löschen eines Bildes höchstens 1/10 s betragen. Denn erst bei derart schnellen Bildwechseln nimmt das Auge die Bildfolge als mehr oder weniger flüssige Bewegung dar. Bei Filmen z.B. werden 24 Bilder in 1 Sekunde präsentiert.

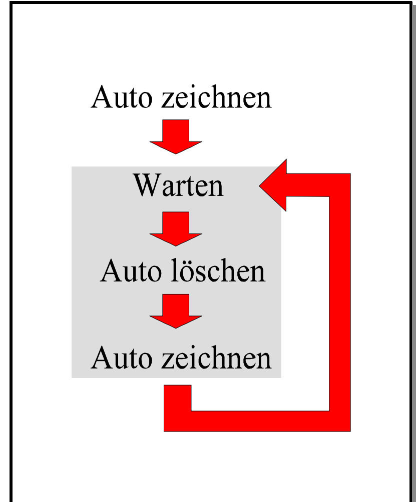



**Abb. 25**: Das Animationsprinzip

Es bleiben nunmehr zwei Fragen zu klären: Wie wird die Wartezeit programmiert? Wie löscht man das Auto?

Um das Programm eine gewisse Zeit warten zu lassen, benutzen wir die Igel-Anweisung **`i.warte(zeit)`**; dabei wird der Parameter `zeit` in ms angegeben. Will man z.B. eine Wartezeit von 1/10 s einlegen, so schreibt man `i.warte(100)`.

Zum Löschen des Autos bedient man sich eines Tricks: Wir zeichnen das Auto an derselben Position einfach noch einmal, allerdings in der Farbe des Hintergrundes. Die Hintergrundfarbe findet man schnell mit einigen Versuchen heraus. (Sie hängt von den Systemeinstellungen des Rechners ab.) Die Funktion zum Löschen des Autos lautet daher:

```
function auto_loeschen()
{
     i.setzeFarbe(191, 191, 191); // meine Löschfarbe
     auto_zeichnen();
     i.setzeFarbe(0, 0, 0); // Farbe wieder auf schwarz!
}
```

***Aufgaben***

1. Programmiere die Animation aus Abb. 24.

2. Programmiere ein sich drehendes Flügelrad (vgl. Aufg. 11 aus dem letzten Abschnitt).

3. Denke dir selbst eine einfache Animation aus und realisiere sie.