

7 Algorithmen

In diesem Kapitel will ich einige einfache Algorithmen mit ihren Kontrollstrukturen in Java besprechen. Zum Nachschlagen steht am Ende dieses Kapitels eine Zusammenfassung aller wichtigen Kontrollstrukturen.

7.1 Zahlensysteme

Du wirst nun einige Algorithmen kennen lernen, die eine Zahl aus dem Dezimalsystem in ein anderes Stellenwertsystem umwandeln.

7.1.1 Zweiersystem

Übung 7.1.1:

- Mache dich vertraut mit dem Dualsystem.
- Wandle um ins Dualsystem: 17_{10} , 27_{10} , 255_{10}
- Wandle um ins Dezimalsystem: 10010_2 , 11101_2 , 101001_2

Du benötigst nun einen geeigneten Algorithmus, mit dessen Hilfe sich eine Zahl aus dem 10-System in das 2-System umwandeln lässt. Abbildung 7.1 zeigt einen solchen Algorithmus als Struktogramm.

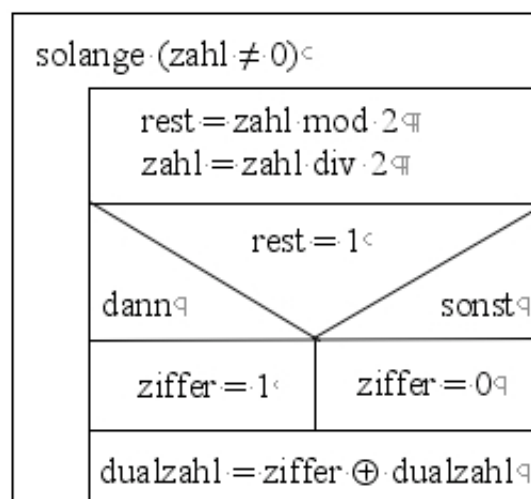


Abbildung 7.1: Struktogramm

Übung 7.1.2:

Studiere den Algorithmus, der in Abbildung 7.1 dargestellt wird. Das Zeichen \oplus bedeutet „anhängen“, z.B. $1 \oplus 3 = 13$ oder $7 \oplus 6 = 76$

Wandle die Zahl 11 vom 10-System ins 2-System um. Trage dazu in der folgenden Tabelle die entsprechenden Werte ein:

rest	zahl	ziffer	dualzahl
...
...

Der in Abbildung 7.1 dargestellte Algorithmus zur Umwandlung einer Zahl aus dem 10-System in das 2-System lässt sich durch folgende Rechnung darstellen:

$$\begin{array}{l}
 11 : 2 = 5 \text{ rest } 1 \\
 5 : 2 = 2 \text{ rest } 1 \\
 2 : 2 = 1 \text{ rest } 0 \\
 1 : 2 = 0 \text{ rest } 1 \quad \Rightarrow \quad 11_{10} = 1011_2
 \end{array}$$

Übung 7.1.3:

Versuche nun, den im Struktogramm dargestellten Algorithmus in Java umzusetzen. Entwerfe den Quelltext auf einem Blatt Papier. Beachte, dass du nicht ein komplettes Programm erstellen musst, sondern nur die Berechnung der Dualzahl, also im Prinzip nur die Methode *berechne()*.

Nach diesen Vorarbeiten kannst du nun die Klasse *Dual* entwerfen, die die Umwandlung durchführt.

Sie benötigt die Datenfelder

zahl speichert die Zahl im 10-System,
dualzahl speichert die umgewandelte Zahl im 2-System.

Weiterhin benötigt sie die Methoden:

holeZahl() besorgt sich die 10-Zahl,
berechne() wandelt diese um ins 2-System,
gibDualzahl() gibt die 2-Zahl wieder zurück.

Ein Modell der Klasse *Dual* könnte also aussehen wie in Abbildung 7.2 dargestellt. Der im Struktogramm dargestellte Algorithmus muss in der Methode *berechne()* implementiert werden.

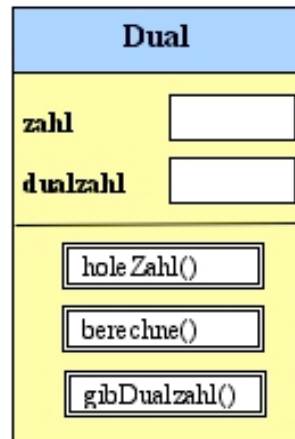


Abbildung 7.2: Modell der Klasse **Dual**

Die Variablen *rest* und *ziffer* sind keine Datenfelder der Klasse *Dual*. Sie werden nur als Zwischenspeicher in der Methode *berechne()* benötigt. Folglich definierst du sie auch dort als so genannte lokale Variablen.

Dieses Modell lässt sich nun sehr einfach in Java umsetzen. Formal hat die Klasse *Dual* also folgendes Aussehen:

```
public class Dual
{
    private int zahl;
    private String dualzahl;

    public Dual()
    {
        ... Konstruktor ...
    }

    /** Holt sich die Zahl, die umgewandelt werden soll. */
    public void holeZahl(int zahlH)
    {
        ... Methodenkörper ...
    }

    /** Wandelt in Dualzahl um. */
    private void berechne()
    {
        int rest;
        String ziffer = "";

        ... Methodenkörper ...
    }

    /** Gibt die Dualzahl aus. */
```

```
public void gibDualzahl()
{
    ... Methodenkörper ...
}
}
```

Abbildung 7.3: Vorläufiger Quelltext der Klasse *Dual*

Übung 7.1.4:

Erstelle in **BlueJ** ein neues Projekt *Dual01*. Implementiere die vorläufige, in [Abbildung 7.3](#) dargestellte Klasse *Dual*. Teste, ob du sie bereits kompilieren kannst.

Die Methoden *holeZahl()* und *gibDualzahl()* sollten kein Problem darstellen.

```
public void holeZahl(int zahlH)
{
    zahl = zahlH;
}

public void gibDualzahl()
{
    System.out.println("Die Zahl lautet: " + dualzahl);
}
}
```

Abbildung 7.4: Die Methoden *holeZahl()* und *gibDualzahl()*

Nun wird der in [Übung 7.1.3](#) erstellte Algorithmus in die Methode *berechne()* implementiert:

```
public void berechne()
{
    int rest;
    String ziffer = "";

    dualzahl = "";
    while (zahl != 0) {
        rest = zahl % 2;
        zahl = (int) (zahl / 2);
        if (rest == 1) {
            ziffer = "1";
        }
        else {
            ziffer = "0";
        }
        dualzahl = ziffer + dualzahl;
    }
}
```

```

    }
}

```

Abbildung 7.5: Die Methode *berechne()*

Übung 7.1.5:

Implementiere alle drei oben besprochenen Methoden. Teste dein Projekt *Dual01*.

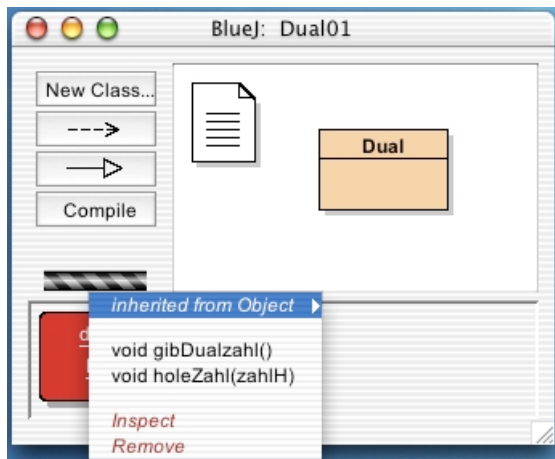


Abbildung 7.6: Projekt *Dual01*



Abbildung 7.7: Ausgabe im Terminal-Fenster

In der Abbildung 7.6 fehlt scheinbar die Methode *berechne()*. Der Grund liegt im Aussehen der grafischen Benutzeroberfläche. Die Abbildung 7.8 zeigt eine Möglichkeit.

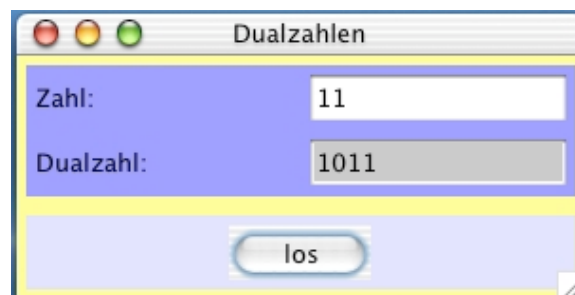


Abbildung 7.8: Die grafische Oberfläche von *Dual03*

Sie zeigt die beiden Datenfelder *zahl* und *dualzahl*. Wird der Button *los* gedrückt, so holt sich ein Objekt der Klasse *Dual* mit *holeZahl()* die 10-Zahl aus dem oberen Textfeld und gibt mit *gibDualzahl()* die berechnete 2-Zahl in das untere Textfeld zurück. Die Umwandlung ist eine interne Sache der Klasse *Dual*, also sollte die Methode *berechne()* als *private* erklärt werden und ist somit

nicht mehr sichtbar, wie die Abbildung 7.6 zeigt. Deswegen wurde deren Aufruf in die Methode *holeZahl()* verlegt.

```
public void holeZahl(int zahlH)
{
    zahl = zahlH;
    berechne();
}
```

Abbildung 7.9: Methode *holeZahl()* ruft die private Methode *berechne()* auf

Übung 7.1.6:

Führe die vorgeschlagenen Veränderungen bei den beiden Methoden *holeZahl()* und *berechne()* durch. Teste dein Projekt *Dual01* ausführlich!

Übung 7.1.7:

Welchen Wert gibt die Methode *gibDualzahl()* zurück, wenn mit Hilfe der Methode *holeZahl()* der Wert 0 eingegeben wird? Lies den Quelltext sorgfältig und versuche den Fehler zu finden.

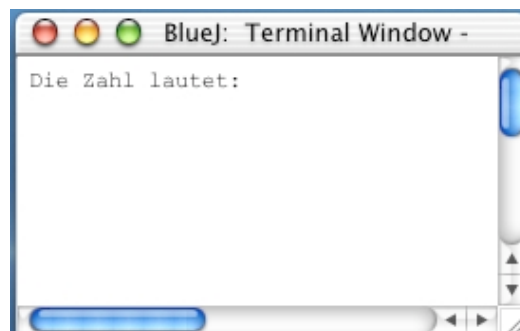


Abbildung 7.10: Das Terminal-Fenster bei der Eingabe von 0₁₀

Ein Softwareentwickler muss die Fähigkeit besitzen, einen Quelltext zu lesen und zu verstehen. Hierbei ist es sehr nützlich, mit Hilfe von zusätzlichen Werkzeugen ein besseres Verständnis davon zu bekommen, was bei einer Programmausführung passiert. Ein Debugger ist ein solches Werkzeug, mit dem ein Entwickler ein Programm Schritt für Schritt ausführen lassen kann. Der Debugger bietet üblicherweise Funktionen zum Stoppen und Starten eines Programms an ausgewählten Stellen im Quelltext und zum Betrachten der Werte von Variablen.

Die Entwicklungsumgebung **BlueJ** bietet einen einfach zu bedienenden Debugger. Klicke dazu in den linken Rand des Editors an die Stelle, an der die Ausführung gestoppt werden soll. Wie in Abbildung 7.11 dargestellt erscheint

ein roter Haltepunkt. Nun rufst du wie gewohnt die Methoden auf. Die Ausführung wird am Haltepunkt gestoppt, das Debugger-Fenster öffnet sich automatisch. Die Kontrollknöpfe am unteren Rand des Fensters können benutzt werden, um die Ausführung des Programms anzuhalten oder fortzusetzen. In Abbildung 7.11 wurde der Haltepunkt in Zeile 33 gesetzt und mit *Step* die nächste Anweisung durchgeführt.

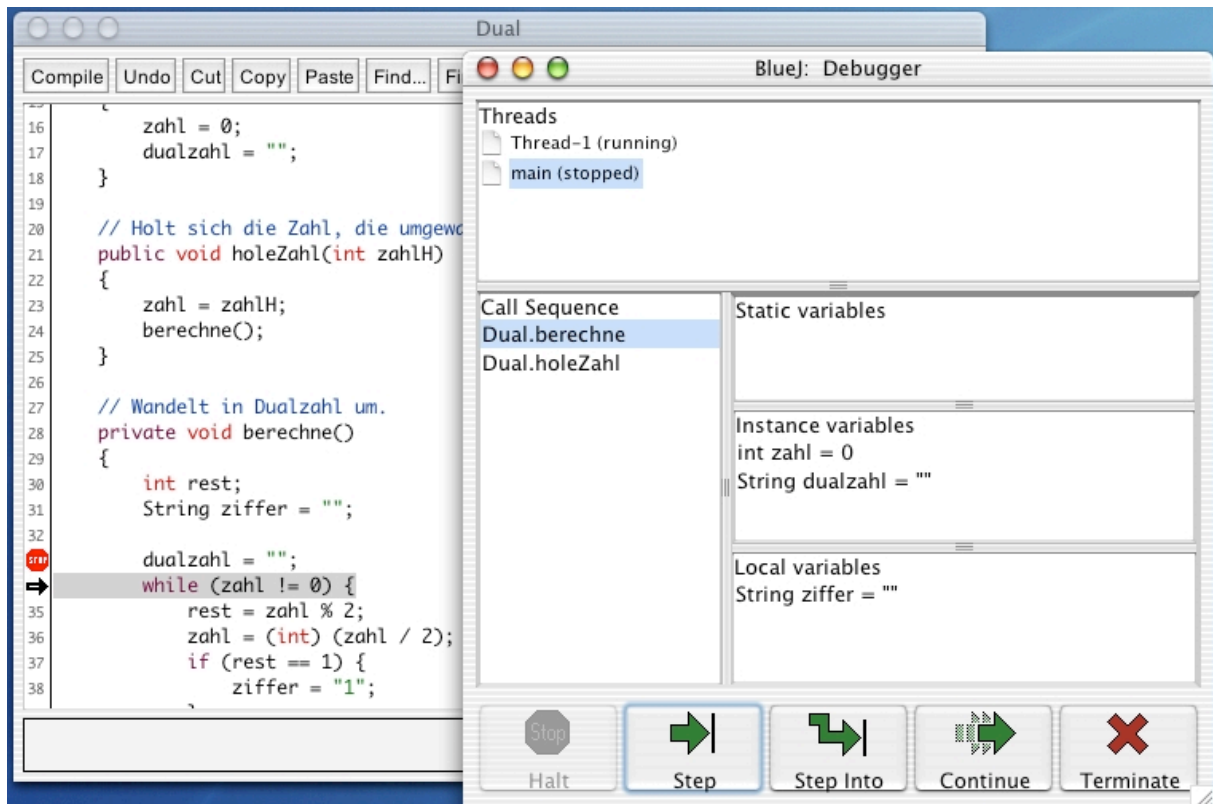


Abbildung 7.11: Der Debugger in BlueJ

Übung 7.1.8:

Untersuche dein Programm mit Hilfe des Debuggers.

Übung 7.1.9:

Erkläre anhand des Struktogramms von Abbildung 7.1, warum im Terminal-Fenster keine Ausgabe der Dualzahl erfolgt.

Zeichne ein neues Struktogramm, damit auch bei der Eingabe von 0_{10} eine Umwandlung in die Dualzahl 0_2 stattfindet.

Übung 7.1.10:

In Kapitel 7.4 (*Zusammenfassung ...*) findest Du eine geeignete Kontrollstruktur. Erstelle ein neues Projekt *Dual02* in **BlueJ** und implementiere eine Klasse *Dual*, die auch die Zahl 0_{10} korrekt in die Dualzahl 0_2 umwandelt.

Übung 7.1.11:

Welchen Wert gibt die Methode *gibDualzahl()* zurück, wenn mit Hilfe der Methode *holeZahl()* der negative Wert (-11) eingegeben wird? Überlege dir eine sinnvolle Fehlerbehandlung.

Übung 7.1.12: (Für Fortgeschrittene und Interessierte)

Erstelle eine grafische Oberfläche und eine stand-alone-Applikation ähnlich wie in Abbildung 7.8 dargestellt. Die dazu notwendige Klasse *Simulation* findest du im Kapitel 4. *Digitaluhr* → *Java ohne BlueJ*. Benenne dein Projekt *Dual03*. Auf dem Schulserver findest du die Datei *DualGUI*, die dir bei der Gestaltung der grafischen Benutzeroberfläche weiterhilft.

7.1.2 Dreiersystem

Nun kannst du auch die Umwandlung einer Zahl aus dem 10-System in ein beliebig anderes System programmieren.

Übung 7.1.13:

Erstelle ein neues Projekt *DreiZahl01*. Ändere die Methode *berechne()*, so dass sie eine Zahl aus dem 10-System in das 3-System umwandelt.

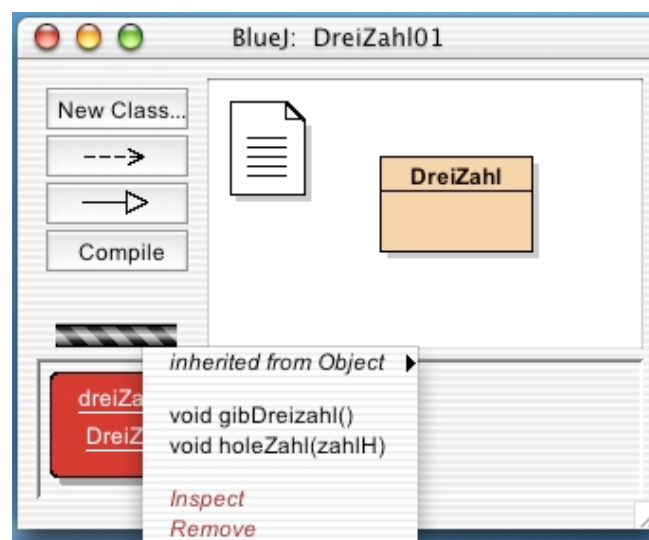


Abbildung 7.12: Projekt *DreiZahl01*

Ein Nachteil sind die vielen *if*-Anweisungen, die den Quelltext schnell unübersichtlich machen.

Übung 7.1.14:

Suche in Kapitel 7.4 (*Zusammenfassung ...*) eine geeignete Kontrollstruktur, die die *if*-Anweisungen ersetzen könnte.

Java bietet eine so genannte Fallunterscheidung in Form einer *switch*-Anweisung an:

```
do {
    rest = zahl % 3;
    zahl = (int) (zahl / 3);
    switch (rest) {
        case 2: ziffer = "2";break;
        case 1: ziffer = "1";break;
        case 0: ziffer = "0";break;
    }
    dreizahl = ziffer + dreizahl;
} while (zahl != 0);
```

Abbildung 7.13: Die *switch*-Anweisung

Übung 7.1.15:

Implementiere die *switch*-Anweisung in der Methode *berechne()*. Teste nun dein Programm.

Übung 7.1.16: (Für Fortgeschrittene und Interessierte)

Erstelle nun ein Programm, das eine Zahl aus dem 10-System ins 16-System (Hexadezimalsystem) überführt.

Beispiel:

$$\begin{array}{l} 1278 : 16 = 79 \text{ rest } 14 (= E) \\ 79 : 16 = 4 \text{ rest } 15 (= F) \\ 4 : 16 = 0 \text{ rest } 4 \end{array} \Rightarrow 1278_{10} = 4FE_{16}$$

7.2 KingKong

In der 5. Klasse wird im Rahmen der Teilbarkeitslehre häufig folgendes Spiel durchgeführt:

Die Spieler bilden einen Kreis. Der erste Spieler fängt an und sagt „eins“, der Spieler rechts neben ihm nennt die nächste Zahl „zwei“. Nun kommt der Spieler rechts neben dem zweiten Spieler an die Reihe. Er darf aber nicht „drei“ sagen, denn anstelle einer Zahl, die durch 3 teilbar ist, muss der Spieler „King“ sagen. Ist die Zahl durch 7 teilbar, so muss der Spieler, der an der Reihe ist, „Kong“ sagen. Manche Zahlen sind durch 3 und auch durch 7 teilbar. Dann muss derjenige Spieler „KingKong“ anstelle dieser Zahl sagen.

Du wirst nun dieses Spiel programmieren. Zuerst wird eine Methode erstellt, die auf die Teilbarkeit durch $king = 3$ überprüft. Hierzu verwendest du die *Modulo*-Anweisung. Sie berechnet den Rest bei einer ganzzahligen Division. Wenn als Rest der Wert 0 berechnet wird, so ist die Zahl durch $king$ teilbar.

```
private void berechne()
{
    int king = 3;

    if (zahl % king == 0) {
        wort = "King";
    }
    else {
        wort = "" + zahl;
    }
}
```

Abbildung 7.14: Quellcode zur Überprüfung auf Teilbarkeit durch $king$

Nun kannst du die Klasse *KingKong* entwerfen.

Sie benötigt die Datenfelder

zahl speichert die zu überprüfende Zahl,
wort speichert den zugehörigen Text.

Weiterhin benötigt sie die Methoden

holeZahl() besorgt sich die Zahl,
berechne() überprüft auf Teilbarkeit,
gibWort() gibt die Zahl oder „King“ zurück.

Ein Modell der Klasse *KingKong* könnte also aussehen wie in Abbildung 7.15 dargestellt.

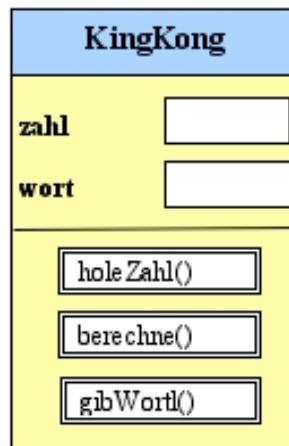


Abbildung 7.15: Modell der Klasse *KingKong*

Übung 7.2.1:

Erstelle das Projekt *KingKong01*. Implementiere die in Abbildung 7.15 dargestellte Klasse *KingKong*.

Die Methoden *holeZahl()* und *gibWort()* kannst du weit gehend aus der Klasse *Dual* entnehmen und dürften keine Schwierigkeiten bereiten.

Übung 7.2.2:

Erstelle das Projekt *KingKong02*. Erweitere die Klasse *KingKong*, so dass sie Teilbarkeit auf 3 oder die Teilbarkeit auf 7 oder die Teilbarkeit auf 3 und zugleich 7 überprüft. Entsprechend sollen die Worte „King“ oder „Kong“ oder „KingKong“ oder die Zahl selbst ausgegeben werden.

Übung 7.2.3:

Erstelle das Projekt *KingKong03*. Nun sollen anstelle der Zahlen 3 und 7 beliebige Zahlen in die Datenfelder *king* und *kong* eingegeben werden können. Ändere die Methode *holeZahl()* um in die gewünschte Methode und bezeichne diese nun mit *holeDaten()*. (Vgl. Abbildung 7.16)

Nachdem du dich mit der Klasse *KingKong* vertraut gemacht hast, wirst du ein Programm entwickeln, das die natürlichen Zahlen bis zu einer oberen Grenze *bis* hinaufzählt. Bei einer Teilbarkeit durch *king* bzw. *kong* sollen die Zahlen ersetzt werden durch die Worte King bzw. Kong.

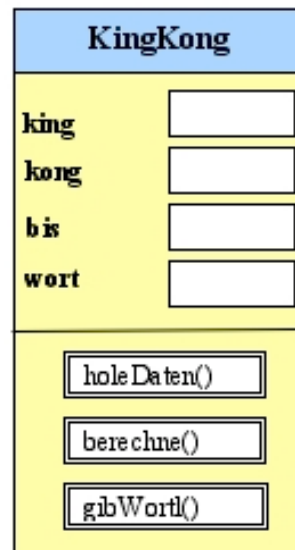


Abbildung 7.16: Modell der Klasse *KingKong*

Das Datenfeld *zahl* wird nun nicht mehr benötigt, neu hinzu kommt dafür die obere Grenze *bis*. Die Methode *holeDaten()* muss entsprechend angepasst werden. Das Hinaufzählen der natürlichen Zahlen übernimmt nun eine Schleife.

```

for (int zahl = 1; zahl <= bis; zahl++) {
    ... Methodenrumpf ...
}
  
```

In den Methodenrumpf schreibst du die bereits in *KingKong03* erstellten Teilbarkeitsregeln. Jedoch musst du hier eine kleine, aber entscheidende Änderung vornehmen. In diesem Projekt soll die Zahlenreihe in der Form

1 2 King 4 5 King Kong 8 King 10 11

dargestellt werden. Das bedeutet, das *wort* wird um die nächste Zahl bzw. Wort erweitert, es werden also am Ende der Zeichenkette weitere Zeichen angehängt. Die Anweisung

```
wort = "KingKong";
```

wird ersetzt durch

```
wort = wort + "KingKong";
```

kürzer durch:

```
wort += "KingKong";
```



Abbildung 7.17: Projekt *KingKong04*

Abbildung 7.17 zeigt die beiden öffentlichen Methoden. Die Methode *holeDaten()* fragt nach den Werten für die Datenfelder *king* und *kong*, sowie nach der oberen Grenze *bis*. Die Methode *gibWort()* ist zuständig für die Ausgabe im Terminal-Fenster.

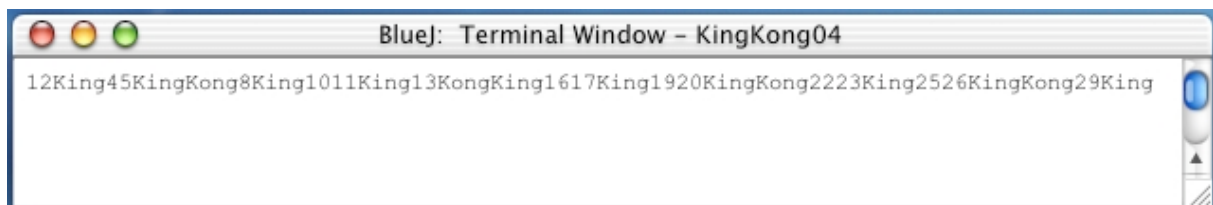


Abbildung 7.18: Das Terminal-Fenster zeigt die Zahlenreihe

Abbildung 7.18 zeigt im Terminal-Fenster die Zahlenreihe für die Werte *king* = 3, *kong* = 7 und *bis* = 30. Auf eine geeignete Darstellung der Zahlenreihe werden wir später eingehen.

Übung 7.2.4:

Erstelle ein Projekt *KingKong04*. Implementiere die oben erwähnten Änderungen. Vergleiche dein Projekt mit den Abbildungen 7.17 bzw. 7.18.

Das letzte Problem ist noch eine geeignete Darstellung der Zahlenreihe. In Abbildung 7.19 ist nach jeder zehnten Zahl ein Zeilenumbruch vorgenommen worden und die einzelnen Zahlen sind durch senkrechte Striche getrennt.



Abbildung 7.19: Ausgabe der Zahlenreihe im Terminal-Fenster

Dies wird erreicht, indem man in der *for*-Schleife der Methode *berechne()* folgende Anweisungen am Ende anfügt:

```
if (zahl % 10 == 0) {  
    wort += "\n";  
}  
else {  
    wort += " | ";  
}
```

Abbildung 7.20: Steuerung des Zeilenumbruchs in der Methode *berechne()*

Übung 7.2.5:

Erstelle ein Projekt *KingKong05*. Implementiere die oben erwähnten Änderungen. Vergleiche die Ausgabe deines Projekts mit der Abbildungen 7.19.

Fortgeschrittene und Interessierte können nun eine grafische Oberfläche entwerfen. Ein Beispiel zeigt die Abbildung 7.21.



Abbildung 7.21: Die grafische Oberfläche von *KingKong06*

Übung 7.2.6: (Für Fortgeschrittene und Interessierte)

Erstelle eine grafische Oberfläche und eine stand-alone-Applikation ähnlich wie in Abbildung 7.21 dargestellt. Auf dem Schulserver findest du die Datei *KingKongGUI*, die dir bei der Gestaltung der grafischen Benutzeroberfläche weiterhilft.

7.3 Zahlenrätsel

In den letzten Beispielen hast du die Textausgaben mit der Anweisung *System.out.println()* in das Terminal-Fenster verlegt. Dieses Terminal-Fenster kann auch für Texteingaben verwendet werden. Jedoch ist die Eingabe von Daten von der Tastatur etwas komplizierter als bei der Verwendung von Textfeldern, die eine grafische Benutzeroberfläche bietet. Im folgenden Projekt *Raetsel* wirst du das Terminal-Fenster zur Eingabe von Daten verwenden.

Es soll eine bestimmte Zahl zwischen *min* und *max* erraten werden. Ist die geratene Zahl *antwort* kleiner als die Lösungszahl *loesung*, so wird der Benutzer aufgefordert, eine größere Zahl einzugeben und umgekehrt. Das Programm gibt also dem Anwender gezielte Hilfestellungen, um die Lösung zu finden. Abbildung 7.22 zeigt einen solchen Algorithmus als Struktogramm.

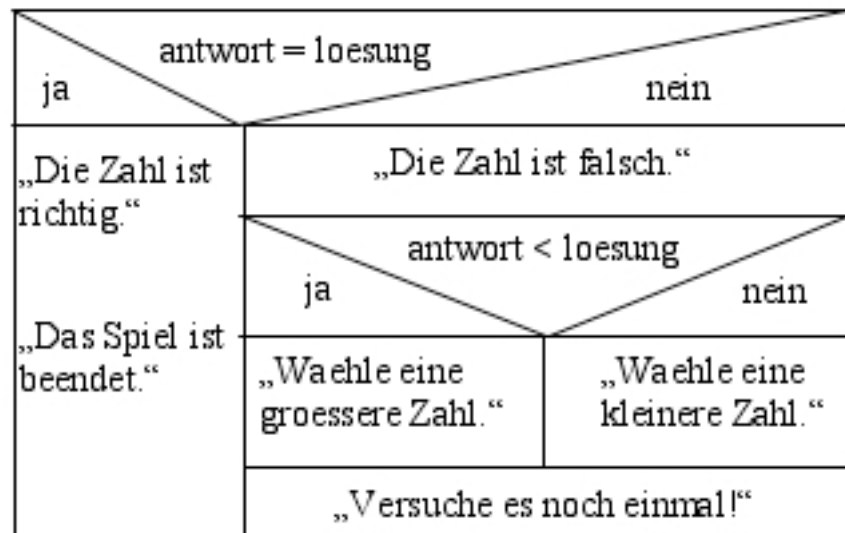


Abbildung 7.22: Struktogramm

Übung 7.3.1:

Studiere den Algorithmus, der in Abbildung 7.22 dargestellt wird. Spiele ihn mit ein paar selbst gewählten Zahlen durch.

Abbildung 7.23: Modell der Klasse **Raten**

Abbildung 7.23 zeigt ein Modell der Klasse *Raten*. Der im Struktogramm dargestellte Algorithmus muss in der Methode *verarbeiteZahl()* implementiert werden.

Übung 7.3.2:

Erstelle in **BlueJ** ein Projekt *Raetsel01*. Implementiere die in den Abbildungen 7.22 und 7.23 dargestellte Klasse *Raten*. Der Konstruktor dieser Klasse soll einem geeignet definierten Anfangszustand herstellen (z.B. *loesung* = 9, *antwort* = 6). Teste dein Projekt und formuliere die Mängel.

Der entscheidende Mangel ist wohl, dass für jede neue Zahl *antwort* der Quelltext verändert werden muss. Mit Hilfe einer Methode *holeZahl()* kann man dem Datenfeld *antwort* einen neuen Wert zuweisen.

Übung 7.3.3:

Erstelle in **BlueJ** ein Projekt *Raetsel02*. Implementiere eine Methode *holeZahl()*. (Hinweis: Solche Methoden hast du bereits in den letzten Beispielen erstellt!) Teste dein Projekt, indem du das Datenfeld *antwort* mit einer Zahl belegst, die kleiner/gleich/größer der *loesung* ist.

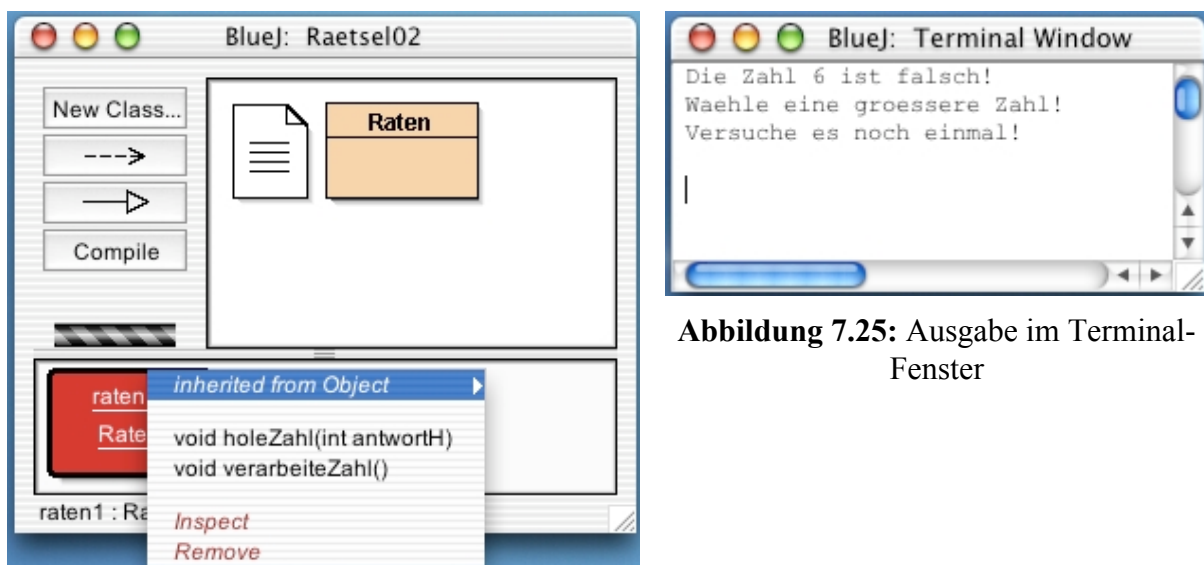


Abbildung 7.25: Ausgabe im Terminal-Fenster

Abbildung 7.24: Projekt *Raetsel02*

Die Abbildungen 7.24 und 7.25 deuten an, dass mit Hilfe der Methode *holeZahl()* in das Datenfeld *antwort* die Zahl 6 eingelesen und mit der *loesung* (hier die Zahl 9) verglichen wurde.

Beim Testen deines Projekts wirst du wohl festgestellt haben, wie störend jedes Mal der Aufruf der Methode *holeZahl()* ist. Der Benutzer erwartet eigentlich, gleich im Terminal-Fenster die nächste Zahl eingeben zu können. Die Methode *holeZahl()* muss also automatisch aufgerufen werden.

Übung 7.3.4:

Deklariere die Methode *holeZahl()* als *private* und rufe diese Methode innerhalb der Methode *verarbeiteZahl()* auf. (Vgl. dazu in der Klasse *KingKong* die Methode *berechne()*!)

Warum bereitet diese Lösungsmöglichkeit Schwierigkeiten?

Du wirst nun eine Möglichkeit kennen lernen, in das Terminal-Fenster mit der Tastatur Eingaben zu machen und diese zu verwerten.

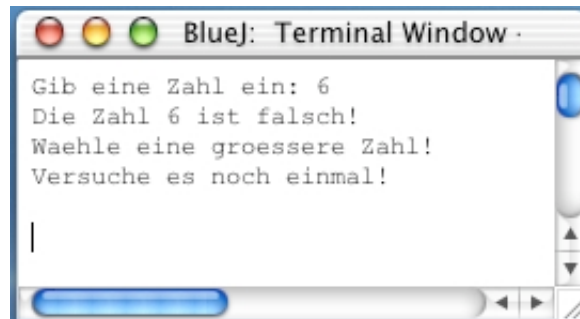


Abbildung 7.26: Eingabe der Zahl 6 in das Terminal-Fenster

Abbildung 7.26 zeigt, dass nach der Aufforderung „Gib eine Zahl ein: „ die Zahl 6 in das Terminal-Fenster eingegeben werden kann. Da diese kleiner ist als die Lösungszahl 9, muss nun eine größere Zahl gewählt werden.

Leider ist die Eingabe von Daten in das Terminal-Fenster komplizierter als die Eingabe in Textfelder von grafischen Benutzeroberflächen. Im *Handbuch der Java-Programmierung* stellt G. Krüger ein kleines Beispielprogramm „Listing0203.java“ auf S. 60f im Kapitel „2.3.3 Einfache Eingabe“ vor.

Übung 2.3.5:

Im Computerraum wirst du das o.g. Buch finden. Ansonsten frage deinen Informatik-Lehrer. Suche das entsprechende Beispiel, implementiere es in **BlueJ** und teste es. Vergleiche deine Ausgabe mit der von G. Krüger vorgeschlagenen Ausgabe.

Versuche das Listing soweit wie möglich zu verstehen.

Du wirst nun den von G. Krüger vorgeschlagenen Quelltext so verändern, dass er sich für unsere Methode *holeZahl()* eignet. Da *holeZahl()* immer noch als *private* deklariert wurde, muss sie in der Methode *verarbeiteZahl()* aufgerufen werden.

```
private void holeZahl()
{
    System.out.print("Gib eine Zahl ein: ");

    BufferedReader eingabe =
        new BufferedReader(new InputStreamReader(System.in));
    try {
```

```
        antwort = Integer.parseInt(eingabe.readLine());
    } catch (java.io.IOException exc) {}
}
```

Abbildung 7.27: Quelltext der Methode *holeZahl()*

Übung 7.3.6:

Erstelle in **BlueJ** ein Projekt *Raetsel03*. Implementiere eine Methode *holeZahl()*. Teste dein Projekt. Welches Problem taucht nun auf?

Beim Testen deines Projekts solltest du nun eine ähnliche Ausgabe im Terminal-Fenster wie in Abbildung 7.26 erhalten. Die Dateneingabe „6“ kann das Programm nun verarbeiten und mit der Lösung „9“ vergleichen. Leider ist es unmöglich, eine weitere Zahl einzugeben. Es müsste wieder eine Aufforderung zur Eingabe einer weiteren Zahl erscheinen, die wiederum verarbeitet werden muss. Dieser Vorgang wiederholt sich solange, bis der Benutzer die korrekte Zahl *loesung* erraten hat.

Am besten erreicht man diese ständige Wiederholung in einer *while*-Schleife. Eine lokale Variable *boolean beendet* wird auf den Wert *false* gesetzt. Wenn der Benutzer die richtige Zahl eintippt, wird *beendet* auf *true* gesetzt und das Spiel ist beendet.

Übung 7.3.7

Ergänze das Struktogramm in Abbildung 7.22 so, dass weitere Zahlen eingegeben werden können.

Übung 7.3.8:

Erstelle in **BlueJ** ein Projekt *Raetsel04*. Implementiere deine Version des Struktogramms in der Methode *verarbeiteZahl()*. Teste dein Projekt. Wird das Spiel mit der richtigen Zahl beendet oder kann man noch eine weitere Zahl eingeben?

Übung 7.3.9:

Zu Beginn des Spiels soll der Benutzer eine kleine Spielanleitung erhalten. Implementiere dazu eine Methode *gibAnleitung()*. Einen Vorschlag zeigt Abbildung 7.28.

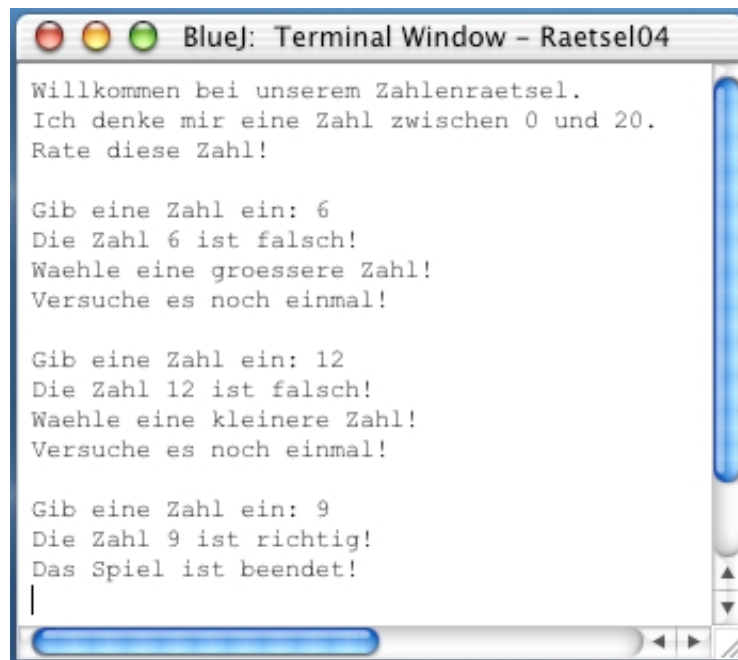


Abbildung 7.28: Ablauf unseres Spiels **Zahlenrätsel**

Übung 7.3.10:

Du sollst nun Zufallszahlen als Lösung erzeugen. Ersetze dazu im Konstruktor die Anweisung

```
loesung = 9;
```

durch

```
loesung = (int) (Math.random() * 20);
```

Übung 7.3.11: (Für Fortgeschrittene und Interessierte)

Programmiere in **BlueJ** ein Spiel *Buchstaben*, in dem die bisherige Suche nach einer Zahl durch eine Suche nach einem Buchstaben ersetzt wird. Abbildung 7.29 zeigt eine mögliche Ausgabe im Terminal-Fenster.

Hinweis: Im „Handbuch der Java-Programmierung“ von G. Krüger werden in Kapitel 11 auf S. 245ff *Strings* beschrieben. Beachte auch die Unterscheidung zwischen Groß- und Kleinbuchstaben.

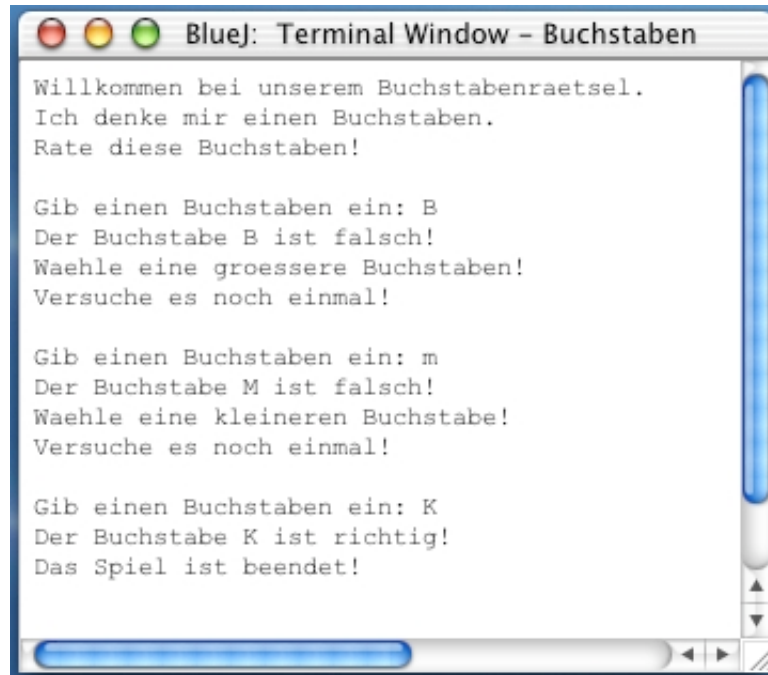


Abbildung 7.29: Ablauf des Spiels **Buchstabenrätsel**

7.4 Zusammenfassung der wichtigsten Kontrollstrukturen

***for*-Schleife**

```
for (Start; Bedingung; Aktualisierung) {  
    Rumpf;  
}
```

Start und Aktualisierung sind Zuweisungen und die Bedingung enthält Variablen, die auch in Start und Aktualisierung vorkommen.

Die Schleife beginnt mit der Ausführung der Start-Anweisung, dann erfolgt die Überprüfung der Bedingung. Ist sie wahr, so wird der Rumpf ausgeführt, dann die Aktualisierung und wieder die Bedingung. Liefert die Bedingung falsch, so wird die Schleife verlassen.

***while*-Anweisung**

```
while (Bedingung) {  
    Anweisungen;  
}
```

Wertet die Bedingung aus. Ist sie wahr, so werden die Anweisungen ausgeführt und die Bedingung dann erneut ausgewertet. Dieser Prozess wird solange wiederholt, bis die Bedingung falsch wird. Dann wird die Ausführung am Ende der Schleife fortgesetzt.

***do*-Anweisung**

```
do {  
    Anweisungen;  
} while (Bedingung)
```

Die Anweisungen werden einmal ausgeführt. Danach wird die Bedingung überprüft. Ist sie wahr, so wird die Schleife wiederholt. Wird die Bedingung als falsch ausgewertet, so endet die Schleife und die nächste Anweisung wird ausgeführt.

***if*-Anweisung**

```
if (Bedingung) {  
    Anweisungen1;
```

```
}  
else {  
    Anweisungen2;  
}
```

Wertet die Bedingung aus. Ist sie wahr, so werden die Anweisungen1 ausgeführt; ist sie falsch, werden die Anweisungen2 ausgeführt.

Der **else**-Teil ist optional. Ist er nicht vorhanden, so führt eine falsche Bedingung dazu, dass die Ausführung mit der Anweisung nach der **if**-Anweisung weitermacht.

switch-Anweisung

```
switch (switch-Ausdruck) {  
    case Wert : Anweisung; break;  
    case Wert : Anweisung; break;  
    . . .  
    default : Anweisung; break;  
}
```

Der **switch**-Ausdruck, der von einem der ganzzahligen Typen oder vom Typ *char* sein muss, wird ausgewertet und sein Wert mit den aufgezählten Werten verglichen. Kommt er dort vor, so beginnt die Ausführung bei der entsprechenden Anweisung und macht mit den folgenden Anweisungen weiter, bis das Ende des **switch** oder ein **break** erreicht wird. Ist der **switch**-Wert nicht in der Liste enthalten, so wird die **default**-Anweisung ausgeführt. Ist auch die nicht vorhanden, so wird die ganze **switch**-Anweisung abgeschlossen, ohne etwas zu bewirken. Jeder Wert darf nur einmal vorkommen.