# Informatik mit Java

Eine Einführung mit **BlueJ** und der Bibliothek **Stifte und Mäuse**

Band 2

Bernard Schriek

# Informatik mit Java

Eine Einführung mit

BlueJ

und der Bibliothek

Stifte und Mäuse

Band II

Nili-Verlag, Werl

Bibliografische Information der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet unter <http://dnb.ddb.de> abrufbar.



bernard.schriek@t-online.de



ISBN-10: 3-00-019637-4
ISBN-13: 978-3-00-019637-9


Bernard Schriek
Ostlandstr. 52
59457 Werl

Geschrieben mit dem Programm Ragtime 6 der Ragtime GmbH, Hilden
<http://www.ragtime.de>
Druck und Verarbeitung: Sächsisches Digitaldruck Zentrum GmbH, Dresden
<http://sdz-directworld.de>
Printed in Germany

# Inhalt

# Danksagung

Das Paket *Stifte und Mäuse (SuM)* wurde in den 90er Jahren im Rahmen der Lehrerfortbildung von Ulrich Borghoff, Dr. Jürgen Czischke, Dr. Georg Dick, Horst Hildebrecht, Dr. Ludger Humbert und Werner Ueding entwickelt. Die Bibliothek wurde zuerst in Object Pascal und Oberon geschrieben und dann auf andere objektorientierte Sprachen portiert. Die Urfassung des SuM-Kern-Pakets in Java wurde mir von Dr. Georg Dick zur Verfügung gestellt. Im Rahmen der Planung einer weiteren Fortbildungsreihe, an der ich auch beteiligt war, wurde die Bibliothek SuM-Kern um weitere Pakete erweitert. Fokke Eschen schrieb die erste Version des Programmgenerators, die dann im Rahmen einer besonderen Lernleistung von Alexander Bissaliev (Abi 2005) neu geschrieben wurde. Viele in diesem Buch behandelten Projekte wurden von den oben genannten Kollegen bei mehreren Fortbildungen zur objektorientierten Programmierung entwickelt und auch im Unterricht erprobt. Umfangreiche methodisch-didaktische Materialien wurden auf dem Bildungsserver Learnline bereitgestellt `<http://www.learnline.de/angebote/oop/>`. Dort gibt es auch ein Forum zum Gedankenaustausch. Der OOP-Bereich auf Learnline wird von Horst Hildebrecht verwaltet. Dieses Buch soll ein weiterer Baustein zum SuM-Paket sein. Ich bedanke mich bei allen Entwicklern dieses Pakets sowie den Kollegen in den verschiedenen Fortbildungsgruppen, die durch anregende Diskussionen die Weiterentwicklung unterstützt haben.

Die freie Java-Entwicklungsumgebung *BlueJ* wurde Ende der 90er Jahre an der Monash-Universität in Australien speziell für den Unterricht entwickelt und auch ständig weiter entwickelt. Besonderer Dank geht an Michael Kölling von der University of Kent, der die BlueJ-Webseiten `<http://www.bluej.org>` pflegt und mit David J. Barnes ein hervorragendes Lehrbuch zur objektorientierten Programmierung mit BlueJ geschrieben hat.

Die Programmiersprache *Java* wurde von Sun Microsystems, Santa Clara, Kalifornien entwickelt. Neben der Objektorientierung und Plattformunabhängigkeit ist die einfache, klare und konsistente Syntax und das konsequente Sprachkonzept ein besonderer Vorzug dieser Programmiersprache, die inzwischen an fast allen Hochschulen und Fachhochschulen eingesetzt wird. Mein Dank gilt allen Javaentwicklern sowie der Firma Sun, die das Java-Paket kostenlos zur Verfügung stellt.

Besonderer Dank geht an Horst Hildebrecht, der das Manuskript kritisch durchsah und mir viele Anregungen für Verbesserungen gab.

Nicht zuletzt gilt mein Dank meiner Frau, die meine stundenlangen Sitzungen am Computer mit viel Geduld ertragen hat und mich immer wieder zur Weiterarbeit ermutigte.

Werl, im September 2006

Bernard Schriek

# Vorwort an Schüler

*Wenn man als Werkzeug nur einen Hammer hat, sieht jedes Problem plötzlich wie ein Nagel aus.*

Abraham Maslov

Nachdem Sie in Band I die Grundlagen der Objektorientierten Programmierung und insbesondere die Ereignisorientierte Programmierung kennen gelernt haben, sollen Sie sich in diesem Band II mit den wichtigsten Datenstrukturen und den zugehörigen Algorithmen beschäftigen.

In der Informatik ist eine Datenstruktur eine bestimmte Art, Daten zu verwalten und miteinander zu verknüpfen, um in geeigneter Weise auf diese zugreifen und diese manipulieren zu können. Datenstrukturen sind immer mit bestimmten Operationen verknüpft, um eben diesen Zugriff und diese Manipulation zu ermöglichen.

Während in Band I meist einfache grafische Beispiele behandelt wurden, geht es in diesem Band um komplexe und hoffentlich für Sie interessante Probleme. Um diese Probleme zu lösen, sollen Sie sich im Verlauf dieses Kurses die entsprechenden Werkzeuge, nämlich die passenden Datenstrukturen, als Klassen herstellen und in eine eigene Bibliothek integrieren. Hierbei gilt das Dualitätsprinzip: *Je besser die Datenstruktur dem Problem angepasst ist, desto einfacher werden die Algorithmen zur Problemlösung*.

Im Laufe der Zeit werden Sie sich so eine Bibliothek von Strukturen aufbauen, deren Klassen Sie bei neuen Problemen immer wieder benutzen werden. Dazu müssen die Strukturen aber möglichst allgemein gehalten werden.

Ein Beispiel für ein solches Vorgehen haben Sie in Band I schon kennen gelernt: die Ereignisanwendung. Sie ermöglicht eine andere, nämlich eine *anwenderorientierte* Sichtweise auf Programme. Alle in diesem Buch behandelten Programme arbeiten ereignisorientiert.

In den verschiedenen Kapiteln werden Sie ausgehend von einer konkreten Problemstellung die dazu passenden Datenstrukturen entwickeln und weitestgehend verallgemeinern. In späteren Kapiteln wird dann auf diese Strukturen zurückgegriffen.

Insbesondere werden lineare Strukturen und Baumstrukturen mit den zugehörigen Standardalgorithmen behandelt. In vielen Programmbibliotheken, auch den von Sun mitgelieferten Java-Klassen, finden Sie diese Strukturen mit den entsprechenden Operationen wieder. Hier geht es aber darum, diese Strukturen *selbst* zu entwickeln, um ihre Vor- und Nachteile besser abschätzen zu können.

Dieses Buch will Ihnen aber auch einige grundlegende Algorithmen zum Suchen und Sortieren mitgeben. Sie helfen dabei, Techniken der Programmierung *abzuschauen*. Denn durch Üben und durch das Studium von Beispielen begreift man ein neues Fachgebiet am schnellsten. Programmieren macht da keine Ausnahme, es ist wie Auto fahren oder Hochseilartistik hauptsächlich eine Sache der Übung.

In den letzten beiden Kapiteln werden Sie mehrere unterschiedliche Verfahren zur Lösung von komplexen Problemstellungen kennen lernen. Sie werden sehen, wie ähnliche Verfahren bei ganz unterschiedlichen Problemen benutzt werden können. Es ist natürlich eine sehr anspruchsvolle Aufgabe die zur Problemstellung passende Strategie zu finden.

Weitere Beispiele für interessante Probleme der Informatik, die Sie auch sehr gut für Facharbeiten nutzen können, finden Sie im Internet beim Problem der Woche zum Informatikjahr: <http://www.informatikjahr.de/index.php?id=193>

In diesem Buch lernen Sie die eigentliche Aufgabe der Informatik kennen: die übersichtliche und effiziente Verwaltung von Informationen und die Verarbeitung dieser Informationen.

Die Beispiele und Aufgaben wurden in mehreren Jahren vom Autor und mehreren anderen Lehrern im Unterricht erprobt und von mehreren Schülern vor der Drucklegung getestet. Wenn Sie aber Verbesserungsvorschläge zu machen haben oder Fehler finden, schreiben Sie eine Email an

bernard.schriek@t-online.de

Viel Erfolg bei Ihrer Informatikausbildung

Bernard Schriek

Werl, im September 2006

# Vorwort an Lehrer

Dieses Buch versucht Lehrerinnen und Lehrer im Informatikunterricht zu unterstützen. Viele Lehrkräfte haben ihre erste Unterrichtserfahrung mit Pascal bzw. vor langer Zeit mit Basic gesammelt. In den 80er Jahren wurde mit großem Aufwand die strukturierte Programmierung mit Pascal unter der Informatiklehrerschaft eingeführt. Das Leitmotiv war damals: Lösen von Problemen durch Zerlegung in Teilprobleme. In den 90er Jahren gelang der objektorientierten Programmierung, die mit der Programmiersprache Smalltalk ein Nischendasein geführt hatte, mit der Verbreitung von Borland Delphi und Sun Java der Durchbruch. Insbesondere *Java* wurde zur Standardprogrammiersprache an den Universitäten und Fachhochschulen. Jetzt ist das neue Leitmotiv: Objekte schicken sich Botschaften und reagieren darauf. Die Konsequenz ist eine neue Form des Softwareentwurfs. Dazu stellt die Unified Modeling Language (UML) die passenden Diagrammformen zur Verfügung. Zur Dokumentation liefert die Java-Entwicklungsumgebung das passende Werkzeug gleich mit: JavaDoc. Die bei vielen Entwicklern unbeliebte Testphase wird durch spezielle Hilfsmittel zum automatischen Testen unterstützt. In diesem Buch wird großer Wert auf Entwurfstechniken, Dokumentation und Tests gelegt.

Lange Zeit musste der Informatikunterricht mit Werkzeugen arbeiten, die zur professionellen Softwareentwicklung gedacht waren. Inzwischen wurde mit *BlueJ* eine Java-Entwicklungsumgebung speziell für den Unterricht geschaffen. Neben einem leistungsstarken Debugger gibt es jetzt die Möglichkeit interaktiv Objekte zu erzeugen, diese zu inspizieren und ihnen Nachrichten zu schicken. BlueJ wird inzwischen an vielen Schulen und Hochschulen für die Ausbildung der Informatikstudenten genutzt.

Speziell für den Schulunterricht wurde die Bibliothek *Stifte und Mäuse* entwickelt. In ihr ist eine Turtlegrafik integriert. Die in Java komplizierte Ereignisverwaltung wurde in fertige Klassen integriert. Zur Bibliothek entstanden im Rahmen von Lehrerfortbildungen zahlreiche vielfach erprobte Unterrichtsprojekte, die in diesem Buch behandelt werden. Der vorliegende Band II behandelt die Einführung der wichtigsten Datenstrukturen (Felder, linearen Listen, Stapel, Bäume, Hashtabellen). Aber auch Rekursion, Sortierverfahren sowie die Breiten- und Tiefensuche mit Backtracking werden behandelt. Band III handelt von der bei den Schülerinnen und Schülern sehr beliebten Netzwerkprogrammierung, relationalen Datenbanken (SQL) sowie von Grundlagen der Automatentheorie. Band III erscheint im Herbst 07.

An mehreren Stellen im Buch gibt es Hinweise auf Referatsthemen. Schüler können sich dazu die passenden Informationen im Internet suchen.

Zur Unterstützung der Lehrkräfte gibt es eine CD-ROM mit Musterlösungen, die gegen Nachweis der Unterrichtstätigkeit angefordert werden kann.

Die aktuellen SuM-Bibliotheken können aus dem Internet geladen werden.
<http://www.mg-werl.de/sum/> bzw. <http://www.nili-verlag.de/sum/>

Methodisch didaktische Erläuterungen zum Konzept von "Stiften und Mäusen" und vielen behandelten Projekten finden Sie auf dem Bildungsserver *Learnline*. Dort gibt es auch ein Forum zum Gedankenaustausch.
<http://www.learnline.de/angebote/oop/>

# Kapitel 1 Installation der Bibliotheken

BlueJ ist eine Entwicklungsumgebung für Java, die an der Monash University in Australien mit dem Ziel entwickelt wurde, Schülern und Studenten den Einstieg in die objektorientierte Programmierung (OOP) zu erleichtern. BlueJ ist also eine Lernumgebung und nicht dazu gedacht, professionelle Javaprogramme zu erstellen. Dazu gibt es geeignetere Werkzeuge wie zum Beispiel Eclipse <http://www.eclipse.org>. In den folgenden Kapiteln werden Sie die besonderen Möglichkeiten von BlueJ kennen und schätzen lernen.

Auch die SuM-Bibliothek (SuM = Stifte und Mäuse) wurde konzipiert um das Erlernen der OOP zu vereinfachen. Das SuM-Konzept wurde in den 90er Jahren von einer Gruppe von Lehrern aus Nordrhein-Westfalen entwickelt, um den Wechsel von der bis dahin üblichen imperativen Programmierung zur objektorientierten Programmierung zu unterstützen. Für verschiedene Programmiersprachen (Object-Pascal, Delphi, Java und Python) wurden die entsprechenden Bibliotheken erstellt. Im Rahmen mehrerer Lehrerfortbildungen wurden eine Reihe von Projekten erarbeitet, um die Schüler in die verschiedenen OOP-Konzepte einzuführen. Weitere ausführliche Informationen zu SuM finden Sie unter <http://www.learnline.de/angebote/oop>.

Um mit SuM unter BlueJ Javaprogramme entwickeln zu können, müssen zuerst drei Pakete auf ihrem Computer installiert sein:
• BlueJ, die Entwicklungsumgebung
• JDK 1.4.2 oder höher (z.B. 1.5.6), das Java Development Kit und die API-Dokumentation (JDK 1.5.x wird empfohlen)
• die SuM-Bibliotheken, Werkzeuge und Hilfetexte.

## 1.1 Installation von BlueJ

Die aktuelle Version von BlueJ können Sie sich von der BlueJ Website <http://www.bluej.org/download/download.html> herunterladen.

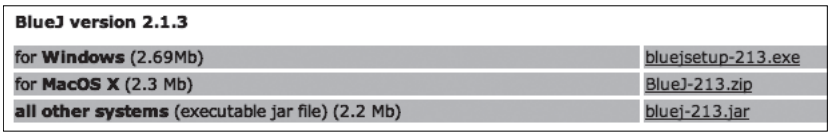


**Abbildung 1.1:**
Download von BlueJ

Für Windows erhalten Sie einen Installer, mit dem Sie BlueJ an einen Ort ihrer Wahl installieren können. Damit allerdings später bestimmte Hilfefunktionen problemlos funktionieren, sollte BlueJ **auf der Festplatte C: im Ordner Programme** installiert werden. Benutzer mit Apple Macintosh und OSX sollten den BlueJ-Ordner **in den Ordner Programme** legen. Starten Sie BlueJ jetzt noch nicht.

## 1.2 Installation des JDK und der API-Dokumentation

Als Nächstes muss das aktuelle Java Development Kit (JDK) auf dem Computer installiert werden. Sie sollten aber vorher überprüfen, ob sich dieses Paket schon auf Ihrem Computer befindet. Dazu öffnen Sie bei Windows Computern die Eingabeaufforderung (das DOS-Fenster). Beim Macintosh öffnen Sie das Terminal. Geben Sie jetzt den Befehl `java -version` ein und betätigen Sie die Returntaste. Falls das JDK installiert ist, erhalten Sie eine Meldung über die installierte Javaversion. Sie sollte mindestens die Versionsnummer 1.4.2 besitzen. Falls Sie allerdings eine Fehlermeldung erhalten, müssen Sie das JDK erst noch auf ihrem Rechner installieren. Windowsbenutzer können sich das JDK von <http://java.sun.com/javase/downloads/index.jsp> herunterladen (J2SE = Java 2 Standard Edition) und installieren. Beachten Sie, dass Sie `JSE 1.4.2` oder höher auswählen (und nicht J2EE = Java 2 Enterprise Edition). Macintoshbenutzer können sich das aktuelle JDK von <http://www.apple.com/java/> herunterladen. Normalerweise ist aber das JDK schon unter OSX vorinstalliert.

| Windows Platform - J2SE and NetBeans IDE Bundle NB 5.0 / J2SE 5.0 Update 8 | | |
|---|---|---|
| J2SE and NetBeans(TM) IDE Cobundle (NB 5.0 / J2SE 1.5.0 Update 8) for Windows | jdk-1_5_0_08-nb-5_0-win-ml.exe | 139.38 MB |

**Abbildung 1.2:**
Download von Java (J2SE SDK)

Für beide Plattformen (Mac und Windows) benötigen Sie jetzt noch die Dokumentation der Javastandardbibliotheken, also die API-Dokumentation (API = Application Programming Interface). Diese finden Sie im gleichen Fenster, von dem aus Sie das JDK geladen haben, etwas weiter unten
<http://java.sun.com/javase/downloads/index.jsp>.

| Multi Platform - J2SE(TM) Development Kit Documentation 5.0 | | |
|---|---|---|
| J2SE(TM) Development Kit Documentation 5.0, English | jdk-1_5_0-doc.zip | 44.05 MB |

**Abbildung 1.3:**
Download der Dokumentation zu Java

Nach dem Download erhalten Sie einen Ordner, dem Sie den Namen `docs` geben und den Sie in den BlueJ-Ordner legen.

## 1.3 Installation der SuM-Bibliotheken

Die aktuellen SuM-Bibliotheken finden Sie unter <http://www.mg-werl.de/sum/> bzw. <www.nili-verlag.de/sum/>. Wenn Sie die zip-Datei runter geladen und entpackt haben, erhalten Sie einen Ordner `SuMWin 6.x` bzw. `SuMMac 6.x`. Dessen Inhalt muss an bestimmte Stellen kopiert werden:

- Den Ordner `doc` legen Sie in den BlueJ-Ordner.
- Den Ordner `docs` legen Sie in den BlueJ-Ordner.

Windows-Benutzer öffnen jetzt den Ordner `lib` im BlueJ-Ordner.

Mac-Benutzer machen einen Rechtsklick bzw. ctrl-Klick auf das Programm BlueJ im BlueJ-Ordner und wählen im Kontextmenü `Paketinhalt zeigen`.
Doppelklicken Sie `Contents - Resources - Java`.

- Ersetzen Sie den Ordner `german` durch den Ordner mit dem gleichen Namen.
- Ersetzen die Datei `bluej.defs` durch die Datei mit dem gleichen Namen.
- Legen Sie `SuMGenerator.jar` in den Ordner `extensions`.
- Legen Sie die folgenden acht jar-Dateien in den Ordner `userlib`: `SuMKern.jar`, `SuMEreignis.jar`, `SuMWerkzeuge.jar`, `SuMKomponenten.jar`, `SuMNetz.jar`, `SuMSql.jar`, `SuMStrukturen.jar`, `SuMMultimedia.jar`.

Schließen Sie die Ordner und starten Sie BlueJ. Falls Sie mehrere JDKs auf Ihrem Rechner installiert haben, werden Sie gefragt, mit welchem JDK Sie arbeiten wollen.

## 1.4 Test der Installation

Als Nächstes sollen Sie kontrollieren, ob alle Komponenten korrekt installiert wurden.

Öffnen Sie die BlueJ-Einstellungen und wählen Sie `Bibliotheken`. Im unteren Teil des Fensters sollte dann stehen, dass die acht SuM-Bibliotheken geladen wurden. Schließen Sie die Einstellungen und wählen Sie im Hilfemenü `Dokumentation sum.kern`. Jetzt sollte sich der Internet-Browser öffnen und die Dokumentation zum Paket `sum.kern` anzeigen. Schließen Sie das Browserfenster. Falls die Dokumentationen nicht angezeigt werden, haben Sie den BlueJ-Ordner nicht an die oben angegebenen Stelle gelegt oder der Ordner `doc` ist nicht im BlueJ-Ordner.

Wählen Sie im Hilfemenü `Java Klassenbibliotheken`. Im Internetbrowser sollte jetzt die Javadokumentation aus dem Ordner `docs/api/index.html`, der im BlueJ-Ordner liegt, angezeigt werden. Falls dies nicht der Fall ist, können Sie sich im Einstellungsfenster von BlueJ unter `Diverses` bei der Eigabe `URL der Javadokumentation` zur oben stehenden Seite durchklicken. Schließen Sie das Browserfenster.

Wählen Sie im Menü `Werkzeuge` den `SuM-Programmgenerator`. Jetzt sollte sich ein Fenster öffnen, in dem Sie verschiedene Komponenten anlegen und das zugehörige SuM-Programm erzeugen können.

Stellen Sie unter `Einstellungen - Editor - Zeilennummern anzeigen` an.

Damit ist die Installation beendet.

# Kapitel 2 Rekursion

**In diesem Kapitel sollen Sie lernen:**

- was Rekursion bedeutet
- wozu die Abbruchbedingung bei der Rekursion dient
- wie man einen Dienst rekursiv aufruft
- wie mehrfache Rekursion funktioniert
- wie Rekursion durch Iteration ersetzt wird
- wann Rekursion besser nicht benutzt werden sollte
- wie der Datentyp `int` im Speicher repräsentiert wird und welche Probleme dadurch auftreten können
- wie man die Klasse `BigInteger` benutzt
- was man unter einem StackOverflow versteht
- wie man die Zählschleife (for-Schleife) als dritte Schleifenform benutzt

*Eine Person ist ein Nachkomme von Dschingis Khan, wenn sein Vater Dschingis Khan ist oder einer seiner Elternteile Nachkomme von Dschingis Khan ist.* Dies bezeichnet man als eine rekursive Definition. Nachkomme von Dschingis Khan wird durch den Begriff Nachkomme von Dschingis Khan definiert.

Ein Dienst heißt *rekursiv*, wenn er durch sich selbst definiert ist.

In diesem Kapitel werden Sie an vielen Beispielen rekursive Dienste kennenlernen. Sie werden sehen, wie elegant Probleme rekursiv gelöst werden können, aber auch Beispiele kennenlernen, bei denen der rekursive Ansatz versagt. Sie werden zuerst einige Beispiele aus der Mathematik kennen lernen, anschließend werden Sie Zeichenketten (Strings) mit rekursiven Diensten manipulieren. Danach werden sie Grafiken rekursiv erzeugen und am Beispiel der "Türme von Hanoi" sehen, wie man ein komplexes Problem mit Rekursion elegant löst. Zum Schluss sollen sie dann noch die Ackermann-Funktion kennen lernen, eine Funktion, die die Grenzen der Berechenbarkeit mit Computern aufzeigt. In den danach folgenden Kapiteln wird immer wieder Rekursion benutzt werden, um Aufgaben zu erledigen. Da bei rekursiven Berechnungen oft sehr große Zahlen auftreten ist im ersten Abschnitt eine Betrachtung des Datentyps `int` und der Klasse `BigInteger` eingefügt.

## 2.1 Fakultät rekursiv und iterativ

Unter der Fakultät versteht man folgende Funktion, die für nichtnegative ganze Zahlen definiert ist:

```
public int fakultaet(int pZahl)
{
   if (pZahl == 0)
```

```
        return 1;
    else
        return pZahl * fakultaet(pZahl -1);
}
```

**Übung 2.1** Berechnen Sie ohne Computer auf dem Papier `fakultaet(5)`.

Die Fakultät der nichtnegativen ganzen Zahl n wird in der Mathematik mit n! (gelesen n-Fakultät) bezeichnet.

Um die Fakultätsberechnung etwas genauer zu untersuchen sollen Sie jetzt mit dem Programmgenerator die Oberfläche für ein Testprogramm erzeugen. Dazu rufen Sie in BlueJ im Menü `Werkzeuge` den Programmgenerator auf.

**Übung 2.2** Erzeugen Sie mit dem Programmgenerator eine Oberfläche wie in Abbildung 2.1. Sie benötigen dazu Etiketten zur Beschriftung, Knöpfe zu Steuerung und einen Zeichenbereich zur Ausgabe des Ergebnisses. Ergänzen Sie die KnopfGeklicktBearbeiter-Dienste und testen Sie das Programm.

Ein *Zeichenbereich* ist ein grafisches Element, in dem in mehreren Zeilen etwas eingegeben oder ausgegeben werden kann. Wenn die Ausgabe an den rechten Rand kommt, erfolgt automatisch ein Zeilenumbruch.

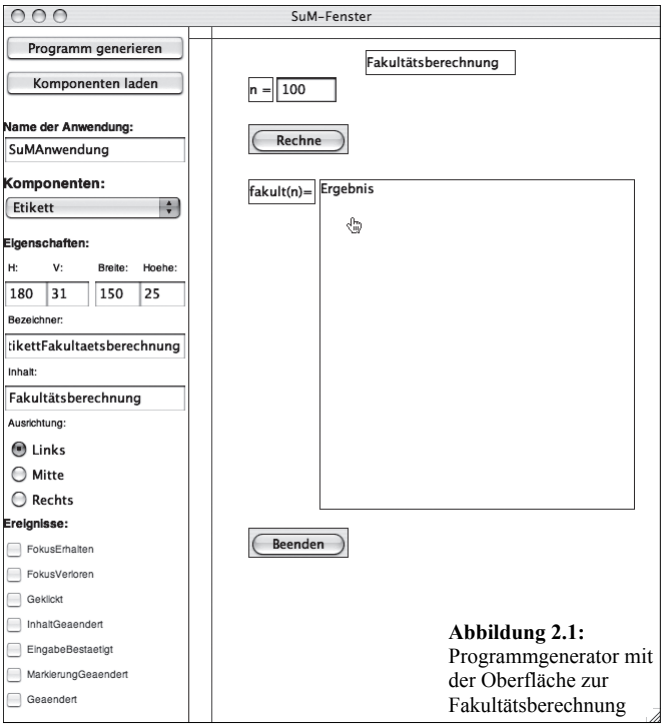


**Abbildung 2.1:** Programmgenerator mit der Oberfläche zur Fakultätsberechnung

Für die Ausgabe wurde ein Zeichenbereich gewählt, damit auch größere Fakultäten in mehreren Zeilen angezeigt werden können.

Beim Test des Programms werden Sie einige seltsame Ergebnisse feststellen:

- Bei der Eingabe von Zahlen bis 31 wird die Fakultät korrekt berechnet.
- Bei der Eingabe von 32 und 33 ist das Ergebnis negativ.
- Bei der Eingabe von Zahlen größer als 33 wird 0 ausgegeben.

Das liegt an der Art, wie Java Daten vom Typ *int* speichert. Jede `int` belegt im Hauptspeicher des Computers 4 Byte, das sind 4 * 8 = 32 Bit. Jedes Bit stellt eine 0 oder 1 dar. Im 5. Schuljahr haben Sie schon mal mit Dualzahlen im Binärsystem gerechnet. Hier bietet sich ein kleiner **Schülervortrag** über Dualzahlen an. Ausführliche Informationen finden Sie unter `<http://de.wikipedia.org/wiki/Dualsystem>`.

Von den 32 Bit einer `int` wird das erste Bit für das Vorzeichen (0 = +; 1 = -) benutzt. Für die eigentliche Zahl bleiben also 31 Bit übrig. Damit kann man dann $2^{31}$ = 2 147 483 648 Zahlen darstellen. Der Zahlbereich für den Datentyp int geht also von -2 147 483 648 bis zu +2 147 483 647. Der positive Bereich ist um 1 kleiner, da die 0 mit zu den positiven Zahlen gerechnet wird. Wenn man also eine sehr große `int` mit einer Zahl multipliziert, kann es sein, dass das Ergebnis größer als die größte positive Zahl wird. Dann wird auch das erste Bit auf 1 gesetzt und in der Anzeige wird die Zahl negativ. Falls die Zahl noch größer wird, setzt die Java-Maschine den Wert auf 0. Ähnliches passiert auch in anderen Programmiersprachen und war schon des öfteren die Ursache für massive Laufzeitfehler in Programmen. In der Informatik wird ein solcher Zahlenüberlauf als *Overflow* bezeichnet. Statt int könnte man den Datentyp `long`, der aus 8 Byte besteht, benutzen. Aber es geht noch besser:

Die Programmiersprache Java bietet für große ganze Zahlen die Klasse `BigInteger` an.

**Übung 2.3** Rufen Sie im Hilfemenü von BlueJ `Java Klassenbibliotheken` auf. Informieren Sie sich über die Klasse `BigInteger`, insbesondere über die Dienste `multiply` und `toString`.

Die Anfrage `fakultaet` lässt sich dann mit Hilfe der Klasse `BigInteger` so umformulieren:

```
public BigInteger fakultaet(int pZahl)
{
   if (pZahl == 0)
            return new BigInteger("1");
   else
            return
              new BigInteger("" + pZahl).multiply(fakultaet(pZahl -
1));
}
```

Der Dienst `hatKnopfRechneGeklickt` muss auch umformuliert werden:

```
public void hatKnopfRechneGeklickt()
{
   hatZeichenbereichErgebnis.setzeInhalt(this.fakultaet(
                        hatTextfeldN.inhaltAlsGanzeZahl()).to-
```

```
String());
}
```

Da der Dienst `setzeInhalt` nur mit den Datentypen `int`, `double`, `char` und `String` funktioniert, muss das Ergebnis der Anfrage `fakultaet` mit Hilfe des Dienstes `toString` vorher in einen String umgewandelt werden.

Um die Klasse `BigInteger` benutzen zu können, muss noch am Anfang des Programms die Zeile

```
import java.math.*;
```

ergänzt werden. Damit lassen sich jetzt größere Fakultäten wie z.B. 1000! berechnen.

**Übung 2.4** Implementieren Sie die Fakultätsberechnung für größere Zahlen mit Hilfe der Klasse `BigInteger`.

Allerdings versagt auch dieses Programm bei 5000! In der BlueJ Konsole erscheint die Fehlermeldung:

```
Fehler in Methode "hatKnopfRechneGeklickt" von Knopf "Rechne":
java.lang.StackOverflowError
```

Es ist ein *StackOverflow* aufgetreten. Dieser Begriff soll jetzt erklärt werden. In Übung 2.1 hatten Sie 5! berechnet:

5! = 5 * 4! = 5 * 4 * 3! = 5 * 4 * 3 * 2! = 5 * 4 * 3 * 2 * 1! = 5 * 4 * 3 * 2 * 1 * 0!
= 5 * 4 * 3 * 2 * 1 * 1 = 120

Dabei wurde die Anfrage `fakultaet` sechsmal aufgerufen. Wenn ein Dienst aufgerufen wird, muss der Computer sich merken, an welcher Stelle im Programmtext das Programm fortfahren soll. In unserem Fall passiert dies also sechsmal. Diese sogenannten Rücksprungadressen werden im Hauptspeicher an einer besonderen Stelle gespeichert, dem *Stack*, der auch *Stapel* oder *Kellerspeicher* genannt wird. Wenn dieser Kellerspeicher voll ist, so dass keine weiteren Rücksprungadressen mehr gespeichert werden können, so entsteht ein Stackoverflow.

**Hinweis** Achten Sie bei der Programmierung von Rekursion darauf, dass der rekursive Aufruf nicht öfter als 3500 mal erfolgt, da sonst ein Stackoverflow auftritt.

Natürlich kann man den Dienst `fakultaet` auch nicht-rekursiv mit einer Schleife implementieren. Das nennt man dann *iterativ*:

```
public BigInteger fakultaet(int pZahl)
{
    BigInteger lProdukt; // lokales Objekt
    if (pZahl < 2)
        return new BigInteger("1");
    else
    {
        lProdukt = new BigInteger("1");
        for (int i = 2; i <= pZahl; i++) // Zählschleife
            lProdukt = lProdukt.multiply(new BigInteger("" + i));
        return lProdukt;
```

```
    }
}
```

Falls pZahl > 1 ist, wird ein lokales Objekt `lProdukt` der Klasse `BigInteger` erzeugt. In einer Zählschleife wird diese `BigInteger` der Reihe nach mit allen Zahlen ≤ `pZahl` multipliziert. Dies ist der ideale Fall für eine *Zählschleife*, die immer dann eingesetzt wird, **wenn die Anzahl der Schleifendurchläufe bekannt ist**. Die Syntax lautet:

```
for (Anfangssituation erzeugen; Durchlaufbedingung; Veränderung des
Durchlaufzählers)
```

**Übung 2.5** Implementieren Sie die Fakultätsberechnung iterativ.

Jetzt können Sie auch große Fakultäten berechnen wie z.B. 30000! Wenn Sie die Anzahl der Stellen des Ergebnisses ermitteln wollen, können Sie den Dienst `anzahl` der Klasse `Zeichenbereich` benutzen. Natürlich dauert die Berechnung jetzt ziemlich lange, da sehr große Zahlen sehr oft multipliziert werden müssen. An diesem Beispiel kann man gut erkennen, dass bei großer Rekursionstiefe die Iteration (Berechnung mit Hilfe einer Schleife) der Rekursion überlegen ist.

## 2.2 Fibonacci rekursiv und iterativ

Der italienische Mathematiker *Leonardo Fibonacci* bzw. Leonardo von Pisa wurde um 1170 vermutlich in Pisa geboren und gilt als der erste bedeutende Mathematiker des Abendlandes. Er veröffentlichte unter Anderem eine Untersuchung über die Vermehrungsrate einer Kaninchenpopulation unter idealen Voraussetzungen. Hier bietet sich ein **Referat über Fibonaccizahlen** und ihre Anwendungen in der Biologie, Kunst usw. an. Im Internet finden Sie umfangreiche Informationen zu diesem Thema. Es gibt sogar eine Fibonaccigesellschaft, die regelmäßig eine Zeitschrift mit neuesten Forschungsergebnissen zu Fibonaccizahlen herausgibt.

Die Fibonaccifolge ist eine Folge von natürlichen Zahlen, deren nächstes Folgenglied aus der Summe der beiden vorhergehenden Fibonaccizahlen gebildet wird. Die Folge beginnt mit zweimal der Zahl 1.

Die ersten Folgenglieder lauten also:

1, 1, 2, 3, 5, 8, 13, 21, ...

**Übung 2.6** Bestimmen Sie die ersten 15 Fibonaccizahlen per Hand.

**Übung 2.7** Erzeugen Sie mit dem Programmgenerator eine GUI (Benutzeroberfläche) wie in Abbildung 2.2 und implementieren Sie die Berechnung mit Hilfe einer **rekursiven** Anfrage `fib(int pZahl)`.

**Übung 2.8** Implementieren Sie die Berechnung mit Hilfe einer **iterativen** Anfrage `fib(int pZahl)`. Tipp: Sie benötigen drei lokale Hilfsvariablen, die in der Schleife verändert werden.

Bei der rekursiven Berechnung von Fibonaccizahlen an Positionen größer als 35 dauert die Berechnung unerträglich lange. Woran kann das liegen? Berechnen Sie dazu einmal

fib(5) auf dem Papier rekursiv. Fällt Ihnen etwas auf?

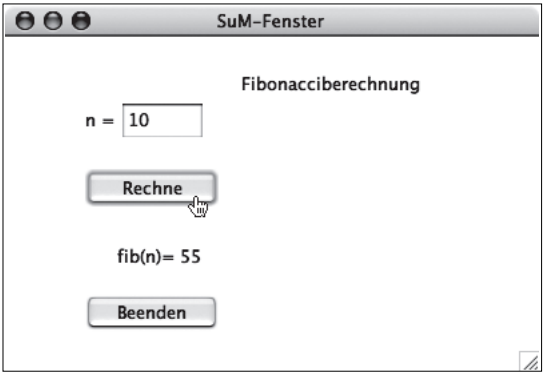


**Abbildung 2.2:** Bildschirmoberfläche zur Fibonacciberechnung

Bei der rekursiven Programmierung der Fibonaccizahlen werden in der Anfrage `fib(n)` die Anfragen `fib(n-1)` und `fib(n-2)` also zweimal `fib` aufgerufen. Dadurch entstehen wieder sehr viele rekursive Aufrufe.

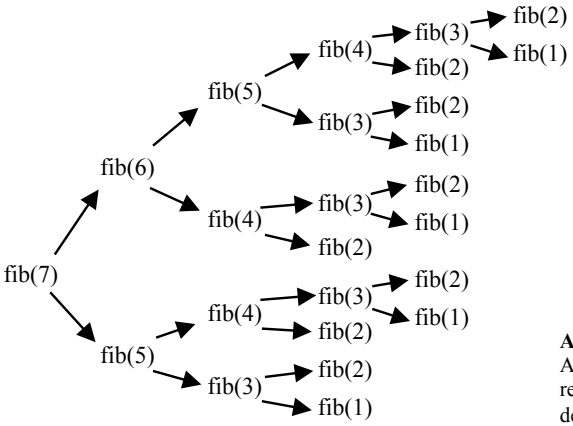


**Abbildung 2.3:** Aufrufbaum bei der rekursiven Berechnung der Fibonaccizahlen

**Übung 2.9** Ergänzen Sie in der Klasse `SuMAnwendung` das Attribut `int zAufrufe`. Dieses Attribut wird im Dienst `hatKnopfBeendenGeklickt` zu Beginn auf 0 gesetzt, zu Beginn der Anfrage `fib` um 1 erhöht und am Ende des Dienstes `hatKnopfBeendenGeklickt` in einem Etikett ausgegeben. So kann man erkennen, wie oft der rekursive Dienst `fib` aufgerufen wurde.

Die Berechnung der Fibonaccizahlen ist ein Beispiel dafür, dass die Berechnung durch Rekursion sehr zeitaufwändig und damit ungünstig ist.

Auch die iterative Berechnung der Fibonaccizahlen liefert schon für n = 47 ein falsches Ergebnis. Hier tritt ein Zahlenüberlauf auf. Die Lösung ist eine Implementierung mit Hil-

fe der Klasse `BigInteger`.

**Übung 2.10** Implementieren Sie die rekursive und iterative Berechnung der Fibonaccizahlen mit Hilfe der Klasse `BigInteger`.

Die *Formel von Binet* liefert eine Funktion, die nur für ganze Zahlen definiert ist (warum wohl?) und für natürliche Zahlen die Fibonaccizahlen liefert:

$$f(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

**Abbildung 2.4:**
Formel von Binet

**Übung 2.11** Implementieren Sie die Berechnung der Fibonaccizahlen mit Hilfe der Formel von Binet. Benutzen Sie dazu die Klasse `Rechner` aus der Bibliothek `sum.werkzeuge`. Beachten Sie beim Testen die Beschränkung auf n < 47 wegen des eingeschränkten Wertebereichs von `int`.

Da die Basis bei der Potenzierung nicht negativ sein darf, muss die Formel umgestellt werden und zwischen geraden und ungeraden Exponenten eine Fallunterscheidung durchgeführt werden.

```
/**
 * Berechnung einer Fibonaccizahl mit der Formel von Binet
 * @param pZahl die Zahl, zu der die Fibonaccizahl berechnet werden
 * soll
 * @return die entsprechende Fibonaccizahl
 */
    public int fib(int pZahl)
    {
        if (pZahl / 2 - 1.0 * pZahl / 2 == 0) // pZahl ist gerade
            return hatRechner.ganzerAnteil(1/hatRechner.wurzel(5)*
                  (hatRechner.potenz((1+hatRechner.wurzel(5))/2,
                   pZahl)-hatRechner.potenz((hatRechner.wurzel(5)-
                   1)/2, pZahl)));
        else // pZahl ist ungerade
            return hatRechner.ganzerAnteil(1/hatRechner.wurzel(5)*
                   (hatRechner.potenz((1+hatRechner.wurzel(5))/2,
                    pZahl)+hatRechner.potenz((hatRechner.wurzel(5)-
                    1)/2, pZahl)));
    }
```

Auch hier ist der Bereich für `pZahl` auf 46 beschränkt, da eine `int` zurückgegeben wird. Für größere Zahlen muss auf die Javaklassen `BigDecimal` und `BigInteger` zurückgegriffen werden.

Im Internet finden Sie beim euklidischen Algorithmus eine Formel, um den größten gemeinsamen Teiler zweier natürlicher Zahlen zu berechnen. Sei a > b:

$$ggt(a,b) = \begin{cases} a & falls\ \ a = b\ \ oder\ \ b = 0 \\ ggt(b,\ a \bmod b) & sonst \end{cases}$$

**Abbildung 2.5:**
Euklidischer Algorithmus

**Übung 2.12** Implementieren Sie den rekursiven euklidischen Algorithmus zur Berechnung des ggT zweier natürlicher Zahlen! Wie kann man das kgV (kleinste gemeinsame Vielfache) berechnen?

## 2.3 Rekursion und Grafik

Als Nächstes sollen Sie rekursive Grafiken entwickeln. Zuerst soll rekursiv eine Folge von immer kürzer werdenden Strichen gezeichnet werden.

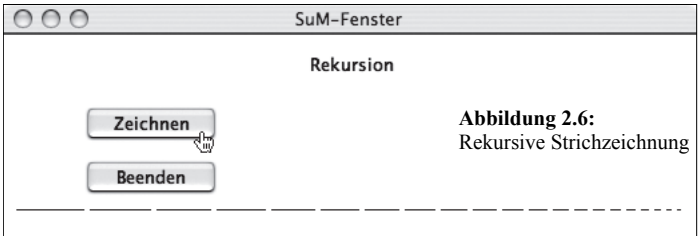


**Abbildung 2.6:**
Rekursive Strichzeichnung

Der rekursive Dienst `zeichneLinie` in der SuMAnwendung lautet:

```
public void zeichneLinie(int pLaenge)
{
    if (pLaenge > 1) // Abbruchbedingung
    {
        hatStift.bewegeUm(pLaenge);
        hatStift.hoch();
        hatStift.bewegeUm(5);
        hatStift.runter();
        this.zeichneLinie(pLaenge * 9 / 10); // rekursiver Aufruf
    }
}
```

**Übung 2.13** Implementieren Sie das Programm zu Abbildung 2.6. Was passiert, wenn der rekursive Aufruf in die erste Zeile der if-Anweisung verschoben wird? Erklären Sie die Änderung in der Zeichnung!

**Übung 2.14** Implementieren Sie ein Programm, welches die Zeichnungen aus Abbildung 2.7 erzeugt. Was passiert, wenn der Parameter beim rekursiven Aufruf `double` statt `int` als Datentyp besitzt?

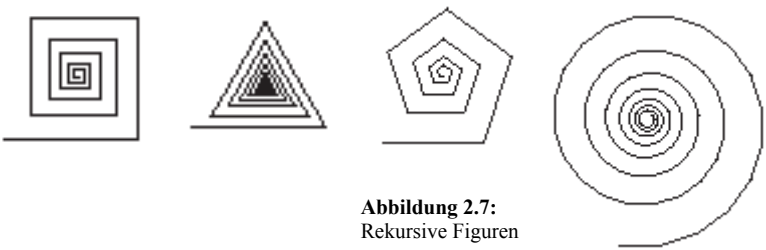

**Abbildung 2.7:**
Rekursive Figuren

**Übung 2.15** (Für leistungsstarke Schüler/innen) Informieren Sie sich im Internet über die Kochkurve und die Hilbertkurve. Schreiben Sie Programme, um diese Kurven zu zeichnen! Sehr bekannt ist auch das Sierpinski-Dreieck.

Jetzt sollen Sie lernen, wie man Bäume und Sträucher rekursiv zeichnet. Sie sollen daran das Prinzip der *Selbstähnlichkeit* kennenlernen.

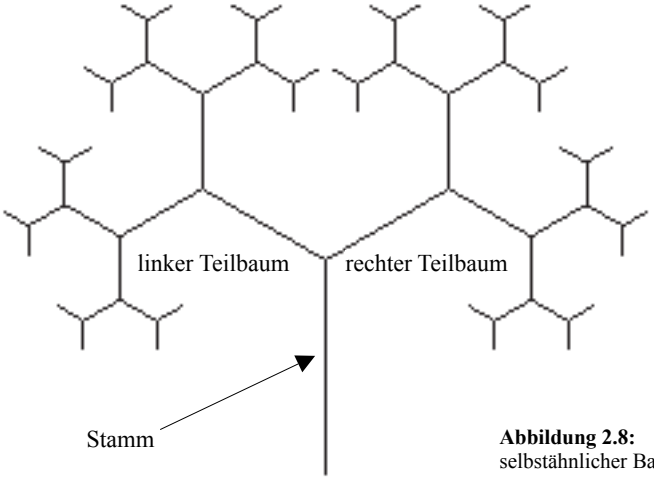


**Abbildung 2.8:**
selbstähnlicher Baum

Der Baum in Abbildung 2.8 besteht aus einem Stamm sowie zwei Teilbäumen. Jeder Teilbaum besteht wieder aus einem Stamm und zwei Teilbäumen, nur kleiner. Da jeder Teilbaum jeweils zwei Teilbäume besitzt, spricht man hier von Binärbäumen (lat. bis = zweimal). Bäume mit jeweils drei Teilbäumen nennt man Ternärbäume usw..

Um solche selbstähnliche Figuren zu zeichnen, eignen sich rekursive Zeichendienste. Iterative Lösungen mit Schleifen würden sehr kompliziert und unübersichtlich. Hier ist die Rekursion der Iteration stark überlegen. Der Baum aus Abbildung 2.8 wird mit folgendem rekursiven Auftrag gezeichnet:

```
public void zeichneBaum(double pLaenge)
{
    if (pLaenge > 10) // Abbruchbedingung
    {
        hatStift.bewegeUm(pLaenge); // Stamm zeichnen
        hatStift.dreheUm(60);
        this.zeichneBaum(pLaenge * 2 / 3); // linker Teilbaum
        hatStift.dreheUm(-120);
        this.zeichneBaum(pLaenge * 2 / 3); // rechter Teilbaum
        hatStift.dreheUm(60);
        hatStift.bewegeUm(-pLaenge); // zurück zum Ausgangspunkt
    }
}
```

Die letzte Anweisung `hatStift.bewegeUm(-pLaenge);` wird oft vergessen. Überle-

gen Sie sich, weshalb sie notwendig ist!

**Übung 2.16** Implementieren Sie ein Programm, um den Baum aus Abbildung 2.8 zu zeichnen! Ändern Sie dann die Winkel und die Größe der Teilbäume! Experimentieren Sie mit unterschiedlichen Winkeln und Größen innerhalb des Auftrags `zeichneBaum`!

**Übung 2.17** Implementieren Sie ein Programm, um Ternärbäume und Quaternärbäume zu zeichnen!

**Übung 2.18** Implementieren Sie ein Programm, bei dem an den Enden des Baums und zwar nur an den Enden kleine rote Kreise (Äpfel) gezeichnet werden (Apfelbaum)!

Nachdem Sie jetzt mathematische Funktionen und Grafiken rekursiv gezeichnet haben, sollen **drei Eigenschaften, die zu jedem rekursiven Dienst gehören**, identifiziert werden:

1.) Es muss eine *Abbruchbedingung* vorhanden sein, damit die Rekursion nicht unendlich weiterläuft. Diese Abbruchbedingung sollte durch eine if-Anweisung am Anfang des Dienstes realisiert werden.

2.) Im Dienst muss sich der Dienst selbst einmal oder mehrmals wieder aufrufen. Das bezeichnet man als den *rekursiven Aufruf*. Falls nur ein rekursiver Aufruf vorkommt, spricht man von einfacher Rekursion, sonst von doppelter, dreifacher usw. Rekursion. Die einfache Rekursion lässt sich leicht durch eine Iteration (Schleife) ersetzen.

3.) Beim rekursiven Aufruf muss sich der Parameter *der Abbruchbedingung nähern*. Meistens ist dies mit der Reduzierung der Komplexität verbunden.

Selbstähnlichkeit kommt in der Natur häufig vor. Dies wird von modernen Grafikprogrammen benutzt, um Landschaften mit Bergen und Seen künstlich zu erzeugen. Besonders bekannt wurde in den 90er Jahren Kai Krause mit seinem Programm Bryce, mit dem man solche selbstähnliche Landschaften erzeugen konnte.

## 2.4 Rekursion bei Strings

Auch Zeichenketten (Strings) lassen sich gut rekursiv bearbeiten. Schauen Sie sich dazu im BlueJ-Hilfemenü unter sum.werkzeuge die Dokumentation der Klasse `Textwerkzeug` an. Insbesondere die Dienste `laenge`, `textOhne`, `textMit`, `verkettung` und `teilzeichenkette` werden benötigt werden.

**Übung 2.19** Welcher String wird bei Eingabe des Strings "INFORMATIK" von der rekursiven Anfrage `wandle`, deren Quelltext unten angegeben ist, zurückgegeben?

```
public String wandle(String pText)
{
   if (hatTW.laenge(pText) > 0)
   {
      return hatTW.verkettung(
               hatTW.teilzeichenkette(pText, 1, 1),
               this.wandle(hatTW.textOhne(pText, 1, 1)),
               hatTW.teilzeichenkette(pText, 1, 1));
```

```
    }
    else
       return ""; // leerer String
}
```

Den erzeugten Text nennt man ein *Palindrom*.

Um die nächsten Aufgaben zu lösen, sollten Sie mit dem Programmgenerator eine Oberfläche (GUI = Graphical User Interface) erzeugen, bei der Sie in einem Textfeld eine Zeichenkette eingeben können, die dann beim Drücken auf einen Knopf mit der rekursiven Anfrage `wandle` gewandelt und in einem Etikett ausgegeben wird. Es ist auch möglich, dafür zu sorgen, dass das Wandeln ausgeführt wird, wenn die Return- oder Enter-Taste im Textfeld gedrückt wird. Dazu müssen Sie im Programmgenerator links unter Ereignisse den Schalter `Eingabe bestätigt` ankreuzen und den entsprechenden Dienst im automatisch erzeugten Programm füllen.

**Übung 2.20** Schreiben und testen Sie einen rekursiven Dienst `wandle`, der alle Buchstaben verdoppelt. Aus JAVA wird dann JJAAVVAA.

**Übung 2.21** Schreiben und testen Sie einen rekursiven Dienst `wandle`, der ein Wort umdreht. Aus JAVA wird dann AVAJ.

**Übung 2.22** Schreiben und testen Sie einen rekursiven Dienst `wandle`, der ein Palindrom erzeugt, bei dem das gedrehte Wort zuerst kommt. Aus JAVA wird dann AVAJJAVA.

**Übung 2.23** Schreiben und testen Sie einen rekursiven Dienst `wandle`, der in einem Wort die Vokale durch einen * (Stern) ersetzt. Aus JAVA wird dann J*V*. Dies kann man erweitern zum Lied "Drei Chinesen mit dem Kontrabass". Wer es nicht kennt, findet Informationen (und ein MP3) im Internet.

Rekursion auf Strings wird in der Praxis bei der Entwicklung von Compilern speziell beim Parsen von Quelltexten intensiv benutzt.

## 2.5 Die Türme von Hanoi

Zum Schluss sollen Sie das wohl berühmteste Beispiel für Rekursion kennen lernen. Vermutlich wurde das Spiel 1883 vom französischen Mathematiker Edouard Lucas erfunden. Er dachte sich dazu die Geschichte aus, dass indische Mönche im großen Tempel zu Benares, im Mittelpunkt der Welt, einen Turm aus 64 goldenen Scheiben versetzen müssten, und wenn ihnen das gelungen sei, wäre das Ende der Welt gekommen.

In der Geschichte lösen die Mönche das Problem folgendermaßen: Der älteste Mönch erhält die Aufgabe, den Turm aus 64 Scheiben zu versetzen. Da er die komplexe Aufgabe nicht bewältigen kann, gibt er dem zweitältesten Mönch die Aufgabe, die oberen 63 Scheiben auf einen Hilfsplatz zu versetzen. Er selbst (der Älteste) würde dann die große letzte Scheibe zum Ziel bringen. Dann könnte der Zweitälteste wieder die 63 Scheiben vom Hilfsplatz zum Ziel bringen.

Der zweitälteste Mönch fühlt sich der Aufgabe ebenfalls nicht gewachsen. So gibt er

dem drittältesten Mönch den Auftrag, die oberen 62 Scheiben zu transportieren, und zwar auf den endgültigen Platz. Er selbst (der Zweitälteste) würde dann die zweitletzte Scheibe an den Hilfsplatz bringen. Schließlich würde er wieder den Drittältesten beauftragen, die 62 Scheiben vom Zielfeld zum Hilfsplatz zu schaffen.

Dies setzt sich nun bis zum 64. Mönch fort, der die Aufgabe, die kleinste und oberste Scheibe zu verschieben, alleine bewältigen kann. Da es 64 Mönche im Kloster gibt und alle viel Zeit haben, können sie das Problem in endlicher, wenn auch sehr langer Zeit lösen.

**Übung 2.24** Spielen Sie das Spiel, indem Sie 4 oder 5 verschieden große Zettel ausschneiden und auf dem Tisch einen Turm bilden, der dann von Position A auf Position B verschoben werden soll, wobei Sie noch eine Position C als Zwischenspeicher benutzen sollen. Notieren Sie die Verschiebungen in der Form: **von A zu B** usw. Gesucht ist die Lösung, die mit der minimalen Anzahl von Verschiebungen auskommt.

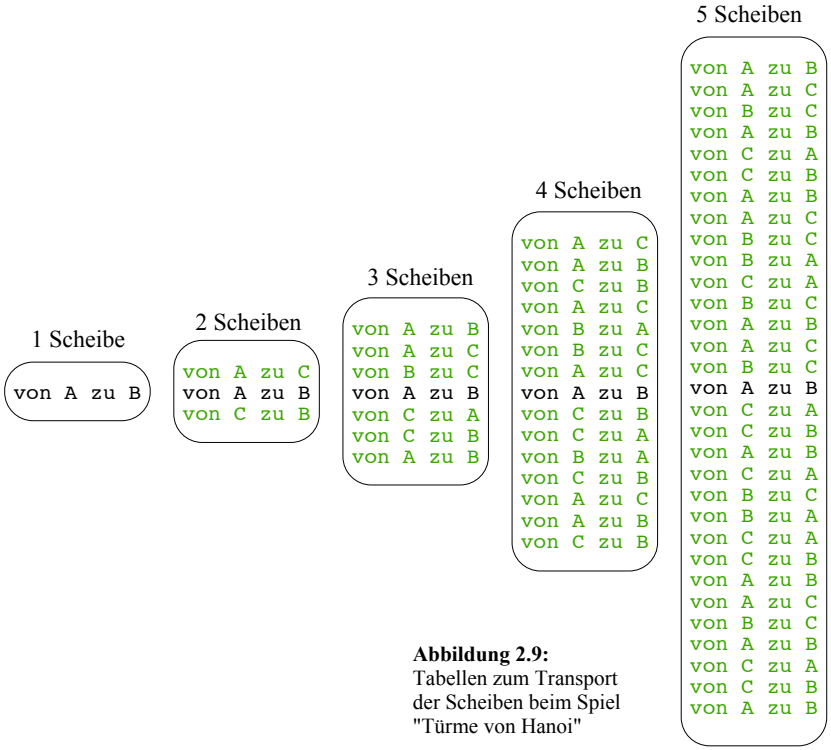


**Abbildung 2.9:**
Tabellen zum Transport der Scheiben beim Spiel "Türme von Hanoi"

Wenn man die Tabellen in Abbildung 2.9 genauer untersucht, erkennt man, dass die Zugfolge aus der linken Tabelle in der rechten Tabelle jeweils zweimal vorkommt. Dies kann mach auch so formulieren:

*Um einen Turm mit n Scheiben von A nach B zu transportieren, muss man einen Turm*

*mit n-1 Scheiben von A nach C transportieren, dann die größte Scheibe von A nach B legen, und zum Schluss den Turm mit n-1 Scheiben von C nach B transportieren.*

Dieser Satz beinhaltet einen Algorithmus mit doppelter Rekursion. Der Transport eines Turmes von n Scheiben wird zurückgeführt auf zweimal den Transport eines Turmes von n-1 Scheiben und dazwischen wird die größte Scheibe umgelegt.

Im Internet finden Sie zahlreiche Darstellungen dieses berühmten Algorithmus, oft mit zugehörigen Animationen. Hier soll der Dienst des Transportierenknopfs sowie der rekursive Dienst `transport` angegeben werden. Die Anzahl der Scheiben wird aus dem Textfeld `hatTextfeldEin` geholt. Die Ausgabe erfolgt in den Zeilenbereich `hatZeilenbereichAus`.

```
public void hatKnopfTransportierenGeklickt()
{
    int n = hatTextfeldEin.inhaltAlsGanzeZahl();
    hatZeilenbereich.loescheAlles();
    this.transportiere(n, 'A', 'B', 'C');
}

public void transportiere(int pAnzahl, char pVon, char pZu,
                          char pMit)
{
    if (pAnzahl > 0)
    {
        this.transportiere(pAnzahl - 1, pVon, pMit, pZu);
        hatZeilenbereichAus.haengeAn("von " + pVon + " zu " + pZu);
        this.transportiere(pAnzahl - 1, pMit, pZu, pVon);
    }
}
```

Um eine grafische Animation des Scheibentransports zu erzeugen, fehlen Ihnen zur Zeit noch Informationen, die Sie im nächsten Kapitel erhalten werden.

## 2.6 Zusammenfassung

In diesem Kapitel haben Sie die Rekursion auf deutsch Selbstbezug kennen gelernt. Rekursion ist die Kunst, ein Problem auf eine einfacheres, aber gleichartiges Problem zurückzuführen.

Rekursive Dienste müssen drei Bedingungen erfüllen.
1.) Es muss eine Abbruchbedingung geben, damit die Rekursion terminiert.
2.) Es muss einen oder mehrere rekursive Aufrufe geben
3.) Das Problem muss beim rekursiven Aufruf *einfacher* sein.

Sie haben Rekursion bei mathematischen Funktionen, bei der Erzeugung von Grafiken und bei der Zeichenkettenverwaltung (Stringverwaltung) kennen gelernt.

Einfache Rekursion lässt sich durch Iteration ersetzen. Bei mehrfacher Rekursion wird das schon deutlich schwieriger. Die Türme von Hanoi waren ein Beispiel dafür, dass sich ein komplexes Problem durch Rekursion elegant lösen lässt.

Manche Programmiersprachen, besonders die sogenannten funktionalen Programmier-

sprachen wie Logo oder Lisp erlauben keine Iteration. Wiederholungen müssen mit Rekursion programmiert werden. Zum Schluss sollen eine rekursive mathematische Funktionen angegeben werden, die in der theoretischen Informatik eine große Bedeutung besitzt, die *Ackermann-Funktion*. Informationen zur Ackermann-Funktion finden Sie im Internet.

```
public int ack(int n, int m)
{
     if (n == 0)
         return m + 1
     else if (m == 0)
         return ack(n - 1, 1)
     else
         return ack(n - 1, ack(n, m - 1))
}
```

## Neue Begriffe in diesem Kapitel

• **Rekursion** (dt. Selbstbezug) Als Rekursion bezeichnet man den Aufruf oder die Definition einer Funktion durch sich selbst. Ohne geeignete Abbruchbedingung geraten solche rückbezüglichen Aufrufe in einen so genannten infiniten Regress (umgangssprachlich Endlosschleife). Einfache Rekursion kann durch Iteration (Anwendung einer Schleife) ersetzt werden. Rekursion ist oft natürlicher und eleganter.

• **rekursiver Aufruf** In einem Dienst wird derselbe Dienst ein- oder mehrmals aufgerufen. Da sich der Computer die Rücksprungadresse merken muss, kann ein StackOverflow auftreten.

• **Fakultät** n! (gelesen n Fakultät) = 1 * 2 * ... * n. Die Fakultät ist eine wichtige Funktion in der Kombinatorik.

• **Fibonaccifolge** Die Fibonaccifolge ist eine Folge von nichtnegativen ganzen Zahlen, wobei das nächste Folgenglied die Summe der beiden vorhergehenden Glieder ist.

• **Binärbaum** Ein Baum, bei dem von jeder Verzweigung genau zwei Äste abgehen heißt Binärbaum.

• **Selbstähnlichkeit** Selbstähnlichkeit ist die Eigenschaft von Gegenständen, Körpern, Mengen oder geometrischen Objekten, in größeren Maßstäben, d.h. bei Vergrößerung dieselben oder ähnliche Strukturen aufzuweisen wie im Anfangszustand. Diese Eigenschaft wird unter anderem von der Fraktalen Geometrie untersucht, da fraktale Objekte eine hohe bzw. perfekte Selbstähnlichkeit aufweisen.

• **Türme von Hanoi** Spiel, bei dem Scheiben nach gewissen Regeln von einem Turm zu einem anderen Turm bei Benutzung eines dritten Turms bewegt werden.

## Java-Bezeichner

• **for-Schleife** Zählschleife, die benutzt wird, wenn die Anzahl der Schleifendurchläufe bekannt ist und während des Schleifendurchlaufs nicht verändert wird.

# Kapitel 3 Felder

**In diesem Kapitel sollen Sie lernen:**

- wie man ein Feld mit primitiven Datentypen benutzt
- wie man ein Feld mit Objekten benutzt
- wie man mehrdimensionale Felder benutzt
- worin die Vorteile und Einschränkungen von Feldern bestehen

Dieses Kapitel ist das erste von einer Reihe von Kapiteln, die sich mit der Verwaltung von *Datensammlungen* (engl. Collections) beschäftigen.

## 3.1 Felder mit primitiven Datentypen

Bei der Wahrscheinlichkeitsrechnung werden oft Würfelexperimente untersucht. Dabei interessiert man sich für die absolute und relative Häufigkeit des Auftretens bestimmter Ergebnisse, z.B. der Augensumme 5 beim Wurf mit zwei Würfeln. Dazu würfelt man z.B. 1000 mal mit zwei Würfeln und zählt, wie oft bestimmte Ergebnisse auftreten. Die Durchführung dieses Zufallsexperiments ist langwierig und langweilig. Ein Javaprogramm erledigt dies schneller. Das Ergebnis soll als Balkendiagramm dargestellt werden.

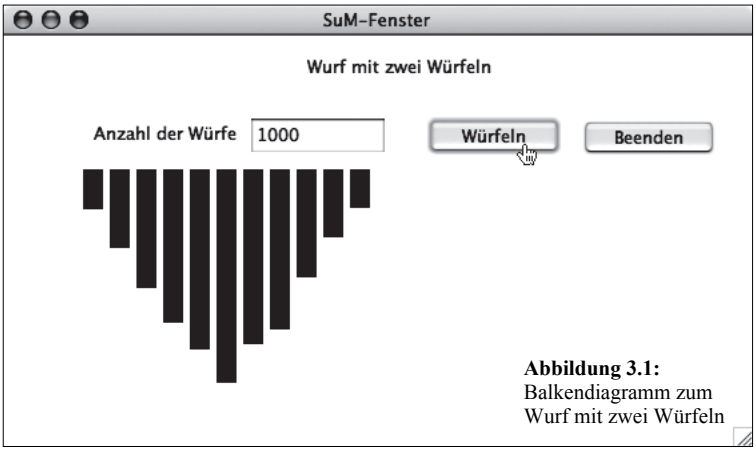


**Abbildung 3.1:** Balkendiagramm zum Wurf mit zwei Würfeln

Bei jedem Wurf werden die beiden Augenzahlen zusammen gezählt und die zugehörige Zählvariable um 1 erhöht. Es gibt insgesamt 11 solcher Zählvariablen vom Typ `int`, da die Augenzahlen 2 bis 12 auftreten können. Diese Zählvariablen kann man in einer Tabelle wie in Abbildung 3.2 darstellen. Wichtig bei dieser Tabelle ist:

- die Zahl der Zeilen steht fest und wird nicht verändert
- in allen Zeilen stehen int-Variablen, also der gleiche Datentyp.

| Augensumme | Anzahl | Index |
|---|---|---|
| 2 | 32 | 0 |
| 3 | 49 | 1 |
| 4 | 73 | 2 |
| 5 | 110 | 3 |
| 6 | 137 | 4 |
| 7 | 168 | 5 |
| 8 | 142 | 6 |
| 9 | 120 | 7 |
| 10 | 82 | 8 |
| 11 | 50 | 9 |
| 12 | 37 | 10 |

**Abbildung 3.2:** Ergebnisse zum Wurf mit zwei Würfeln

Falls eine feste Anzahl gleichartiger Daten (hier `int`) verwaltet werden soll, kann man dazu ein *Feld* (engl. Array) benutzen. Dabei gibt es in Java allerdings eine Besonderheit. Die einzelnen Elemente (Zellen) des Feldes werden durchnummeriert. Diese Nummerierung bezeichnet man als Index. **Dieser Index muss in Java mit 0 beginnen**. Die zu den Augensummen gehörigen Indizes sind in Abbildung 3.2 in der rechten Spalte angegeben. Zur Augensumme 10 gehört also der Index 8. Der zugehörige Programmausschnitt sieht folgendermaßen aus:

```
public void hatKnopfWuerfelnGeklickt()
{
    int lWurf;
    int[] lErgebnis;
    int lAnzahl;
    Buntstift lStift;
    Rechner lRechner;

    lErgebnis = new int[11];
    lAnzahl = hatTextfeldN.inhaltAlsGanzeZahl();
    lStift = new Buntstift();
    lRechner = new Rechner();
    lStift.setzeFuellmuster(Muster.GEFUELLT);
    this.bildschirm().loescheAlles();
    for (int i = 0; i < lAnzahl; i++)
    {
        lWurf = lRechner.ganzeZufallszahl(1, 6)
              + lRechner.ganzeZufallszahl(1, 6);
        lErgebnis[lWurf - 2]++;
        lStift.bewegeBis(20 + 20 * lWurf, 100);
        lStift.zeichneRechteck(15, lErgebnis[lWurf - 2]);
    }
}
```

Das Feld wird mit der lokalen Variablen lErgebnis bezeichnet. Es wird in der zweiten Zeile deklariert: `int[] lErgebnis;` und in der 7. Zeile erzeugt: `lErgebnis = new int[11];`. Das Feld besitzt also 11 Elemente des primitiven Datentyps `int`. Sie erhalten bei der Erzeugung automatisch den Anfangswert 0. Die Variable `lWurf` erhält in der for-Schleife die Augensumme als Inhalt. Überlegen Sie, warum nicht `lRechner.ganzeZufallszahl(2, 12)` benutzt wurde! Anschließend wird mit `lErgebnis[lWurf - 2]++;` der Wert des entsprechenden Feldelements um 1 erhöht. Beachten Sie, dass der Index mit - 2 korrigiert wurde, da die Nummerierung der Feldelemente mit 0 beginnen muss.

**Übung 3.1** Ändern Sie das Programm so ab, dass die Augensumme beim Wurf mit drei Würfeln bestimmt wird! Über den Balken sollen die Augensummen 3, 4 usw. angezeigt werden. Die Balken sollen von unten nach oben wachsen.

Als Nächstes soll ein Feld mit dem primitiven Datentyp `boolean` benutzt werden. Dazu soll der berühmte Algorithmus "*Sieb des Eratosthenes*" programmiert werden. Er dient dazu, Primzahlen zu bestimmen. Die Primzahlen beginnen bei 2. Im Internet finden Sie bei `<http://de.wikipedia.org/wiki/Sieb_des_Eratosthenes>` eine ausführliche Erläuterung des Verfahrens inkl. einer schönen Animation. Beachten Sie bei der Programmierung in Java, dass das boolsche Feld beim Index 0 beginnen muss. Bei der Erzeugung werden die Elemente des Felds automatisch mit `false` initialisiert.

**Übung 3.2** Implementieren Sie das Sieb des Eratosthenes. Es sollen die Primzahlen bis 10000 ermittelt werden und in einem Zeilenbereich ausgegeben werden.

## 3.2 Felder mit Objekten

In einem Ort leben 2500 Menschen, die alle entweder die rote oder die schwarze Partei wählen. Wenn sich zwei Personen treffen, diskutieren sie über Politik und wenn sie unterschiedliche Parteien wählen, gelingt es einem von beiden, den anderen zu überzeugen, in Zukunft die andere Partei zu wählen. Zu Beginn soll ein bestimmter Prozentsatz der Personen die rote Partei wählen. Es stellt sich jetzt die Frage, wie entwickelt sich das Wahlverhalten im Ort. Wird nach einer gewissen Zeit eine Einparteiendiktatur entstehen? Dies soll in einer Simulation untersucht werden. Das Projekt soll `Demokratur` heißen. Damit das Wahlverhalten beobachtet werden kann, sollen die Personen durch rote bzw. schwarze Punkte dargestellt werden, die in einem 50 * 50 Quadrat angeordnet sind. Der Prozentsatz der Roten soll in einem Textfeld eingegeben werden können. Die Programmoberfläche soll wie in Abbildung 3.3 aussehen.

Man erkennt sofort, dass eine Klasse Person benötigt wird. Jede Person wird durch einen Punkt dargestellt, sie benötigt also einen Buntstift, um sich zu zeichnen. Bei der Erzeugung einer Person, wird die Position und die Parteienpräferenz als Parameter übergeben. Die Person benötigt einen Dienst zeichne sowie Dienste zum Abfragen bzw. Verändern der Parteienpräferenz. Der Rechner wird für Zufallszahlen benötigt.

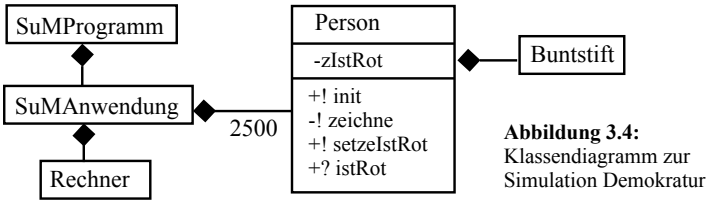


**Abbildung 3.4:**
Klassendiagramm zur Simulation Demokratur

Die 2500 Personen werden in einem **2-dimensionalen Feld** verwaltet. Dieses Feld wird in der Klasse `SuMAnwendung` bei den Bezugsobjekten folgendermaßen deklariert:

```
private Person[][] hatPerson;
hatPerson = new Person[50][50];
```

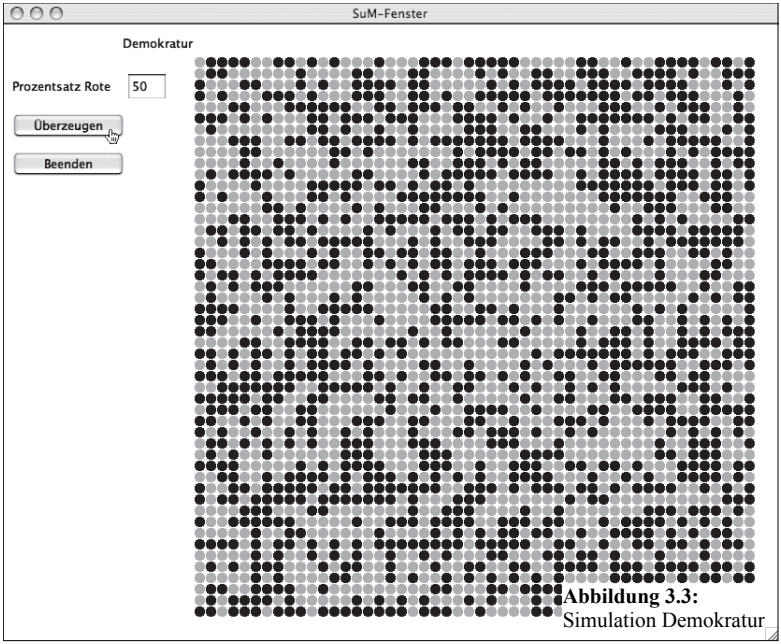


**Abbildung 3.3:** Simulation Demokratur

Durch diese Anweisung wird der Speicherplatz für die 2500 Personen im Hauptspeicher reserviert, die Personen existieren aber noch nicht. Jede Person hat also den Wert `null`. Wenn der Knopf `Überzeuge` gedrückt wird, werden die Personen in einer doppelten for-Schleife erzeugt. Vorher wird eine Zufallszahl `lZufall` zwischen 1 und 100 erzeugt. Falls die Zufallszahl ≤ dem Prozentsatz der roten Parteianhänger ist, wird die neue Person rot sonst schwarz. Dies wird erreicht, indem dem Konstruktor der Person ein entsprechender boolscher Wert als dritter Parameter übergeben wird.

```
public void hatKnopfUeberzeugenGeklickt()
{
    int lZufall; // Zufallszahl, um die Anfangsfarbe zu bestimmen
    int lProzentRot = hatTextfeldRote.inhaltAlsGanzeZahl();
    for (int i = 0; i < 50; i++)
    for (int j = 0; j < 50; j++)
    {
        lZufall = hatRechner.ganzeZufallszahl(1, 100);
        hatPerson[i][j] = new Person(210 + i * 12, 40 + j * 12,
                                     lZufall <= lProzentRot);
    }
    zSimuliert = true;
}
```

Das boolsche Attribut `zSimuliert` wurde im Konstruktor auf `false` gesetzt. Wenn es jetzt im Dienst `hatKnopfUeberzeugenGeklickt` auf `true` geändert wird, kann die Simulation laufen. Dies wird dadurch erreicht, dass im Dienst `bearbeiteLeerlauf` der `SuMAnwendung` der Wert dieses Attributs überprüft wird. Falls der Wert `true` ist, wird ein Simulationsschritt ausgeführt. Da der Dienst `bearbeiteLeerlauf` ständig neu aufgerufen wird, läuft die Simulation, solange `zSimuliert` `true` ist. Man bezeichnet einen

einzelnen Simulationsschritt oft als *Takt*. Ihm entspricht also die Abarbeitung des Dienstes `bearbeiteLeerlauf` in der `SuMAnwendung`. Falls die Simulation zwischendurch mal angehalten werden soll, muss nur `zSimuliert` auf `false` gesetzt werden.

Ein Simulationsschritt besteht aus mehreren Teilaufgaben:

- Wahl einer zufälligen Person
- Wahl eines zufälligen Nachbarn (8 Möglichkeiten)
- zufällige Wahl, wer wen überzeugt
- die eigentliche Überzeugung mit Aktualisierung der Anzeige

```
public void bearbeiteLeerlauf()  // ein Simulationstakt
{
  int l1H, l1V, l2H, l2V;
  // Die Koordinaten der beiden Personen
  int lUeberzeuger; // Wer überzeugt wen?

  if (zSimuliert) // muss ein Takt ausgeführt werden
  {
    // eine Person auswählen
    l1H = hatRechner.ganzeZufallszahl(0, 49);
    l1V = hatRechner.ganzeZufallszahl(0, 49);
    l2H = 0; l2V = 0;
    // Nachbarn auswählen
    switch (hatRechner.ganzeZufallszahl(1, 8))
    {
      case 1: l2H = l1H - 1; l2V = l1V - 1; break;  // links oben
      case 2: l2H = l1H; l2V = l1V - 1; break;      // oben
      case 3: l2H = l1H + 1; l2V = l1V - 1; break;  // rechts oben
      case 4: l2H = l1H - 1; l2V = l1V; break;      // links
      case 5: l2H = l1H + 1; l2V = l1V; break;      // rechts
      case 6: l2H = l1H - 1; l2V = l1V + 1; break;  // links unten
      case 7: l2H = l1H; l2V = l1V + 1; break;      // unten
      case 8: l2H = l1H + 1; l2V = l1V + 1; break;  // rechts unten
    }
    // Testen ob Nachbar existiert
    if (l2H >= 0 && l2H < 50 && l2V >= 0 && l2V < 50)
    {
      // unterschiedliche Farbe?
      if (hatPerson[l1H][l1V].istRot()
          != hatPerson[l2H][l2V].istRot())
      {
        // wer überzeugt?
        lUeberzeuger = hatRechner.ganzeZufallszahl(0, 1);
        // überzeuge den anderen
        if (lUeberzeuger == 0)
          hatPerson[l1H][lPerson1V].setzeRot
              (hatPerson[l2H][l2V].istRot());
        else
          hatPerson[l2H][l2V].setzeRot
              (hatPerson[l1H][l1V].istRot());
      }
    }
  }
}
```

Zum Schluss soll noch der Konstruktor der Klasse `Person` angegeben werden:

```
public Person(int pH, int pV, boolean pIstRot)
{
```

```
  hatStift = new Buntstift();
  hatStift.setzeFuellmuster(Muster.GEFUELLT);
  hatStift.bewegeBis(pH, pV);
  zIstRot = pIstRot;
  this.zeichne();
}
```

**Übung 3.3** Erzeugen Sie mit dem Programmgenerator die Oberfläche zu Simulation und implementieren Sie das Programm. Experimentieren Sie mit verschiedenen Prozentsätzen für die Anzahl der Rotwähler. (`Demokratur1`)

**Übung 3.4** Ändern Sie das Programm so ab, dass bei jedem Takt eine zufällige Person alle 8 Nachbarn von seiner Parteienpräferenz überzeugt. (`Demokratur2`)

**Übung 3.5** Erweitern Sie die beiden Demokraturprojekte so, dass auch die grüne und die gelbe Partei gewählt werden. (`Demokratur3`, `Demokratur4`)

Solche taktgesteuerten Simulationen werden in der Praxis oft benutzt. Bekannt ist eine Räuber-Beute-Simulation mit dem Namen *Wator*. Wenn Sie im Internet nach Wator suchen, finden Sie viele Seiten mit entsprechenden Simulationen. Beachten Sie, dass bei Wator die obere und untere Kante sowie die rechte und linke Kante "zusammengeklebt" sind. Jemand am oberen Rand beeinflusst also auch jemanden am unteren Rand.

## 3.3 Zusammenfassung

Sie haben in diesem Kapitel Felder kennen gelernt. Felder werden benutzt, um eine Menge von gleichartigen Daten zu verwalten. Die **Anzahl der Feldelemente ist fest,** ein entscheidender Nachteil von Feldern. Felder können primitive Datentypen (int, double, boolean, char) oder auch Objekte (im obigen Beispiel `Person`) enthalten. Man spricht die einzelnen Feldelemente über ihren Index an. Der Index wird in eckigen Klammern geschrieben. Felder können auch mehrdimensional sein, wie Sie im letzten Beispiel gesehen haben.

### Java-Bezeichner

• **feld[i][j]** Element eines zweidimensionalen Feldes. i und j sind die Indizes. Die Indizes beginnen in Java immer mit 0. Falls ein Element mit einem ungültigen Index angesprochen wird, löst dies eine IndexOutOfBounds-Exception aus.

### Neue Begriffe in diesem Kapitel

• **Feld** (engl. array) Ein Feld dient zur Verwaltung einer festen Menge gleichartiger Daten. Dabei werden einzelne Feldelemente mit ihrem Index angesprochen. Felder können mehrdimensional sein und primitive Datentypen oder Objekte enthalten.

• **Taktgesteuerte Simulation** (auch diskrete Simulation genannt) ist eine Simulationsart, bei der bei jedem Takt gewisse Aktionen ausgeführt werden. Simulationen können auch durch Ereignisse gesteuert werden.

# Kapitel 4 Lineare Strukturen I - Schlangen

**In diesem Kapitel sollen Sie lernen:**

- wie man Objekte verkettet
- wie man verkettete Objekte in der BlueJ-Werkbank untersucht und manipuliert
- was man unter inneren Klassen versteht und wie man sie implementiert
- was man unter einer generischen Schlange versteht und wie man sie implementiert
- wie man eine eigene Klassenbibliothek erzeugt
- wie man eine eigene Klassenbibliothek in eine Programm einbindet
- wie man eine Schlange auf dem Bildschirm visualisiert

Aus vielen Bereichen des täglichen Lebens kennen Sie Schlangen: beim Stau auf der Autobahn, an der Kasse im Supermarkt und so weiter. Diese Schlangen bezeichnet man in der Informatik als *Warteschlangen*.

In diesem Kapitel soll am Beispiel einer Wartezimmersimulation (Arztpraxis) gezeigt werden, wie man Warteschlangen objektorientiert implementiert.

## 4.1 Verkettung von Objekten

Die Patienten in einer Arztpraxis werden (normalerweise) in der Reihenfolge ihrer Ankunft behandelt. Der Einfachheit halber sollen die Patienten erstmal nur zwei Attribute besitzen: ihren Namen (Meier, Schmidt usw.) und ihre Krankenkasse (AOK, BEK, TKK usw.). Ein jeder Patient kennt seinen Nachfolger, also den Patienten, der als nächster das Wartezimmer betreten hat.

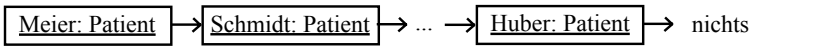


**Abbildung 4.1:**
Patientenschlange

Die Bezeichner in Abbildung 4.1 sind unterstrichen, um anzudeuten, dass die Rechtecke Objekte (nicht Klassen) enthalten.

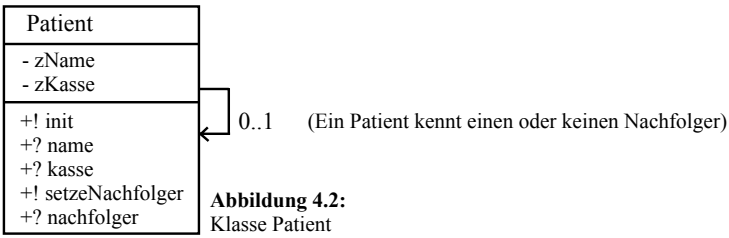


**Abbildung 4.2:**
Klasse Patient

**Übung 4.1** Erzeugen Sie ein neues Projekt `Wartezimmer1` und implementieren Sie die Klasse `Patient` passend zur Abbildung 4.2. Im Konstruktor soll der Name und die Kasse als Parameter übergeben werden. Der Nachfolger soll im Konstruktor auf `null` gesetzt werden.

**Übung 4.2** Erzeugen Sie durch Rechtsklick auf das Klassensymbol drei neue Patienten, die dann unten in der Werkbank als rote Symbole erscheinen sollen. Verketten Sie anschließend diese drei Objekte durch einen Rechtsklick auf die Objektsymbole und Aufruf des Dienstes `setzeNachfolger`. Inspizieren Sie anschließend die Objekte wie in Abbildung 4.4 angegeben. Statt den Objektbezeichner für den Nachfolger mit der Tastatur einzugeben, können Sie auch auf das entsprechende (rote) Objekt klicken.

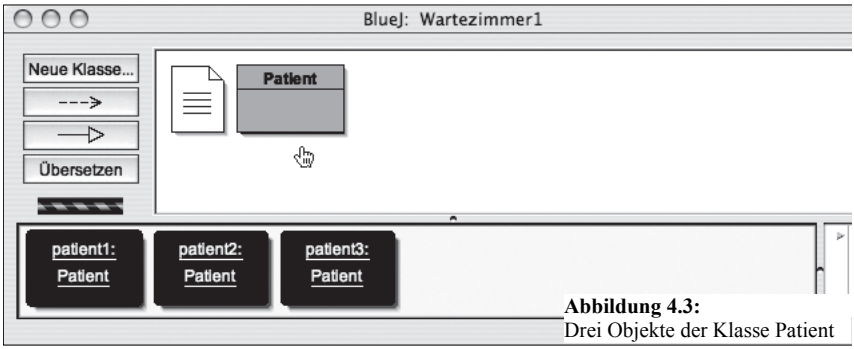


**Abbildung 4.3:**
Drei Objekte der Klasse Patient

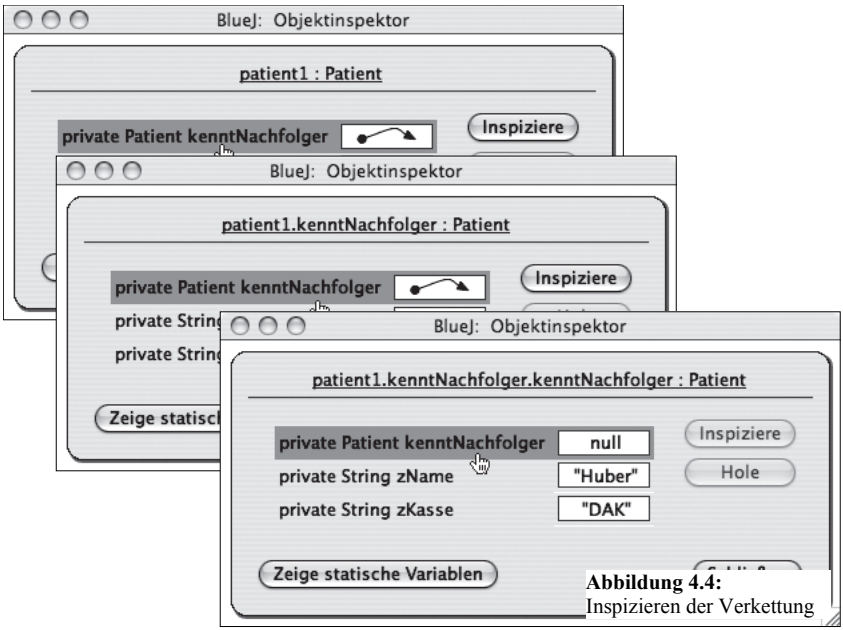


**Abbildung 4.4:**
Inspizieren der Verkettung

## 4.2 Wartezimmersimulation

Bei der Wartezimmersimulation muss man neue Patienten anmelden können. Diese neuen Patienten werden in der Warteschlange hinten angehängt. Der erste Patient (das vorderste Element der Warteschlange) soll angezeigt werden. Außerdem muss es möglich sein, den ersten Patienten zu behandeln, dies soll mit Abmelden (aus dem Warteraum) bezeichnet werden. Die Benutzeroberfläche wird wie immer mit dem Programmgenerator erzeugt und könnte wie in Abbildung 4.5 aussehen. Bezeichnen Sie die Anwendung im Programmgenerator mit `Praxisanwendung`.

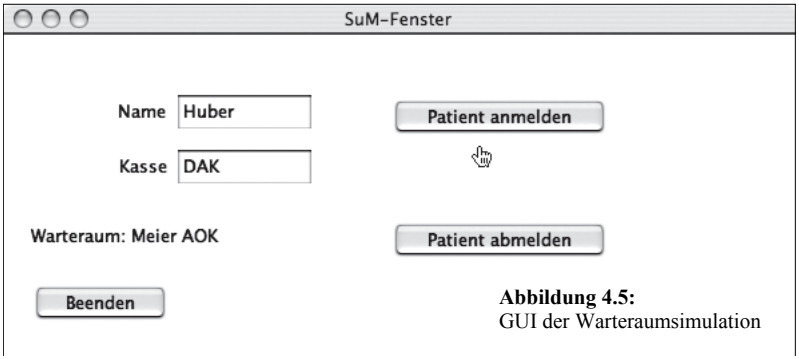


**Abbildung 4.5:**
GUI der Warteraumsimulation

Neben den Etiketten, Textfeldern und Knöpfen der Programmoberfläche (*View*) wird zur Verwaltung der Daten neben der Klasse `Patient` noch die Klasse `Warteraum` benötigt (*Model*). Die Klasse `Praxisanwendung` kontrolliert den Programmfluss (*Controller)*. So wird das MVC-Prinzip gewahrt.

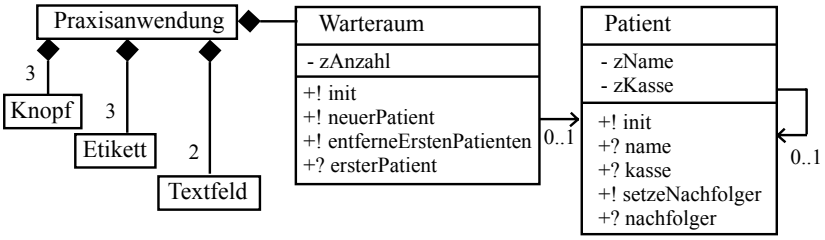


**Abbildung 4.6:**
UML-Diagramm der Warteraumsimulation

Auf den Dienst `gibFrei` in den Klassen `Warteraum` und `Patient` wurde verzichtet, da Java Objekte, die nicht mehr über einen Bezeichner angesprochen werden können, automatisch löscht. Dies bezeichnet man als automatische *Garbage Collection*. In den meisten anderen Programmiersprachen muss sich der Programmierer um die Freigabe der nicht mehr benötigten Elemente selbst kümmern.

Wenn der Benutzer den Knopf `Patient anmelden` drückt, werden die Inhalte der beiden Textfelder ausgelesen und ein entsprechender neuer Patient wird erzeugt. Dieser

neue Patient kann als lokales Objekt `Patient lPatient;` im entsprechenden Dienst deklariert werden. Anschließend wird der Dienst `neuerPatient` des Warteraumobjekts aufgerufen und `lPatient` als Parameter übergeben. Zum Schluss muss dann das Etikett zur Anzeige des Warteraums aktualisiert werden.

```
public void hatKnopfAnmeldenGeklickt()
{
   String lName, lKasse;
   Patient lPatient;

   lName = hatTextfeldName.inhaltAlsText();
   lKasse = hatTextfeldKasse.inhaltAlsText();
   lPatient = new Patient(lName, lKasse);
   hatWarteraum.neuerPatient(lPatient);
   this.zeigeWarteraum();
}
```

Der Warteraum hat ein Bezugsobjekt `kenntErstenPatienten`. Falls kein Patient im Warteraum sitzt, muss `kenntErstenPatienten` auf `null` gesetzt werden. Im Dienst `neuerPatient` muss jetzt eine Fallunterscheidung durchgeführt werden. Wenn der Warteraum leer ist, wird `kenntErstenPatienten` auf den als Parameter übergebenen Patienten gesetzt. Wenn der Warteraum nicht leer ist, wird es komplizierter. Der als Parameter übergebene Patient muss an das Ende der Schlange angehängt werden. Der Warteraum kennt aber das Ende der Schlange nicht, er kennt nur den ersten Patienten.

Dieses Problem wird folgendermaßen gelöst: Es wird ein lokaler Patient `lPatient` deklariert und auf `kenntErstenPatienten` gesetzt. Dann wird in einer Schleife `lPatient` solange auf seinen Nachfolger gesetzt, bis der Nachfolger `null` ist. Jetzt ist also das Ende der Schlange erreicht und `lPatient` ist dieses letzte Schlangenelement. Überlegen Sie bitte, welche Schleifenform ist besser geeignet, while-Schleife oder do-Schleife? Mit dem Dienst `setzeNachfolger` wird dann der neue Patient an `lPatient` angehängt und ist nun das neue letzte Schlangenelement. Zum Schluss muss noch das Attribut `zAnzahl` erhöht werden.

```
public void neuerPatient(Patient pPatient)
{
   Patient lPatient;

   if (kenntErstenPatienten == null)
       kenntErstenPatienten = pPatient;
   else
   {
       lPatient = kenntErstenPatienten;
       while (lPatient.nachfolger() != null)
           lPatient = lPatient.nachfolger();
       lPatient.setzeNachfolger(pPatient);
   }
   zAnzahl++;
}
```

Auch bei der Anzeige des Warteraums in der Klasse `Praxisanwendung` muss eine Fallunterscheidung getroffen werden. Wenn der Warteraum leer ist, soll dies angegeben werden, wenn der Warteraum nicht leer ist, wird der erste Patient angezeigt.

```
public void zeigeWarteraum()
{
```

```
   Patient lPatient;
   lPatient = hatWarteraum.ersterPatient();
   if (lPatient == null)
      hatEtikettNaechster.setzeInhalt("Warteraum ist leer");
   else
      hatEtikettNaechster.setzeInhalt("Warteraum: " +
            lPatient.name() + " " + lPatient.kasse());
}
```

Beim Drücken der Taste `Patient abmelden` wird der Dienst `entferneErstenPatienten` der Klasse `Warteraum` aufgerufen. Hier wird überprüft, ob noch Patienten im Warteraum sind. Ist das der Fall, wird `kenntErstenPatienten` auf seinen Nachfolger gesetzt. Der neue erste Patient ist also der Nachfolger des alten ersten Patienten.

**Übung 4.3** Speichern Sie das Projekt als `Wartezimmer2`. Implementieren und testen Sie die Wartezimmersimulation wie in Abbildungen 4.5 und 4.6 angegeben.

## 4.3 Innere Klassen

In vielen Arztpraxen gibt es aber mehrere Warteschlangen. Es gibt Patienten, bei denen erst der Blutdruck gemessen werden muss, es gibt Patienten, bei denen erst Blut abgenommen werden muss, bevor sie vom Arzt behandelt werden. Hier soll vereinfacht angenommen werden, dass es nur zwei Warteschlangen gibt. Eine Schlange enthält sämtliche Patienten, die andere Schlange enthält diejenigen Patienten, denen erst Blut abgenommen werden muss.

Die Implementierung der Warteschlange im vorigen Abschnitt ist für diese neue Modellierung nicht geeignet, da jeder Patient nur einmal angelegt werden soll. Es bietet sich die Struktur in Abbildung 4.7 an.

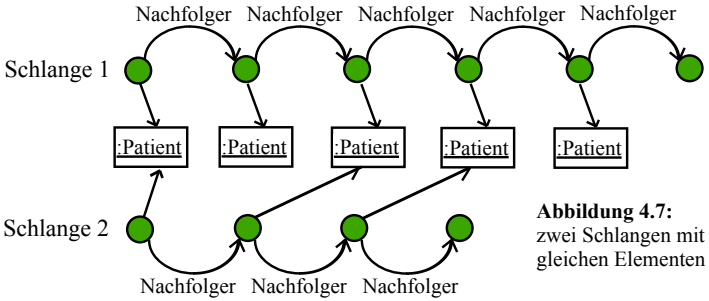


**Abbildung 4.7:** zwei Schlangen mit gleichen Elementen

Die einzelnen Elemente der Schlange sind also nicht mehr Patienten, sie verweisen auf Patienten und nachfolgende Elemente (kennt-Beziehung). Die einzelnen Elemente der Schlange stehen nur der Schlange zur Verfügung. Sie sind also private Objekte der Schlange. Die Programmiersprache Java bietet dazu die Möglichkeit an, innerhalb einer Klasse andere Klassen privat zu definieren. Nach außen sind die zugehörigen Objekte dann nicht sichtbar und können auch nicht direkt angesprochen werden. Die Schlangenelemente sollen in Zukunft als *Knoten* bezeichnet werden, um eine Verwechselung mit den Inhalten (Patienten) zu vermeiden. Das zugehörige UML-Diagramm sieht dann fol-

gendermaßen aus:

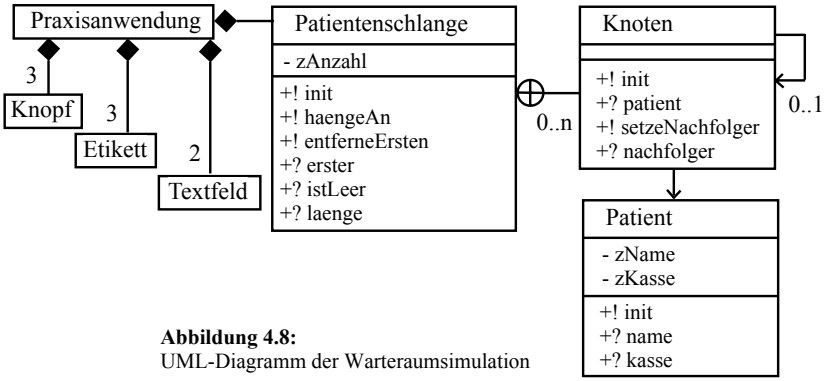


**Abbildung 4.8:**
UML-Diagramm der Warteraumsimulation

Das Wagenradsymbol zwischen den Klassendiagrammen der Klassen `Patientenschlange` und `Knoten` steht für *innere Klasse*. Die Klasse `Patientenschlange` mit ihrer inneren Klasse soll hier angegeben werden.

```
/**
 * @author Bernard Schriek
 * @version 13.04.2006
 */
public class Patientenschlange
{
    // innere Klasse ----------------------------------
    private class Knoten
    {
        // Bezugsobjekte
        private Patient kenntPatient;
        private Knoten kenntNachfolger;

        // Konstruktor
        public Knoten(Patient pPatient)
        {
            kenntPatient = pPatient;
            kenntNachfolger = null;
        }

        // Dienste
        public Patient patient()
        {
            return kenntPatient;
        }

        public Knoten nachfolger()
        {
            return kenntNachfolger;
        }

        public void setzeNachfolger(Knoten pNachfolger)
        {
            kenntNachfolger = pNachfolger;
        }
    }
    // Ende der inneren Klasse ----------------------------
```

```
// Bezugsobjekte
private Knoten kenntKopf;
private Knoten kenntEnde;

// Attribute
private int zAnzahl;

// Konstruktor
/**
 * Eine neue leere Patientenschlange wird erzeugt.
 */
public Patientenschlange()
{
    kenntKopf = null;
    kenntEnde = null;
    zAnzahl = 0;
}

// Dienste
/**
 * Es wird zurückgegeben, ob die Schlange leer ist.
 * @return true, wenn die Schlange leer ist
 */
public boolean istLeer()
{
    return zAnzahl == 0;
}

/**
 * Der erste Patient der Schlange wird zurückgegeben.
 * @return der erste Patient der Schlange
 */
public Patient erster()
{
    if (zAnzahl == 0)
        return null;
    else
        return kenntKopf.patient();
}

/**
 * Ein neuer Patient wird an das Ende der Schlange angehängt.
 * @param pPatient der neue Patient
 */
public void haengeAn(Patient pPatient)
{
    if (this.istLeer())
    {
        kenntKopf = new Knoten(pPatient);
        kenntEnde = kenntKopf;
    }
    else
    {
        kenntEnde .setzeNachfolger(new Knoten(pPatient));
        kenntEnde = kenntEnde.nachfolger();
    }
    zAnzahl++;
}

/**
 * Das erste Element der Schlange wird entfernt.
 */
public void entferneErsten()
```

```
    {
        if (!this.istLeer())
        {
            if (zAnzahl == 1)
            {
                kenntKopf = null;
                kenntEnde = null;
            }
            else
                kenntKopf = kenntKopf.nachfolger();
            zAnzahl--;
        }
    }

    /**
     * Die Länge der Schlange wird zurückgegeben.
     * @return die Anzahl der Elemente in der Schlange
     */
    public void laenge()
    {
        return zAnzahl;
    }
}
```

Beachten Sie das neue Bezugsobjekt `hatEnde`. Es verweist auf das jeweilige Ende der Schlange. So muss beim Anhängen eines neuen Patienten nicht mehr die ganze Schlange vom Kopf bis zum Ende durchlaufen werden. Davon ist der Dienst `haengeAn` sowie der Dienst `entferneErsten` betroffen.

Auch hier soll noch einmal darauf hingewiesen werden, dass es in anderen Programmiersprachen ohne Garbage Collection (z.B. C++ oder Delphi-Pascal) notwendig ist, die Dienste `gibFrei` für die verschiedenen Klassen zu implementieren, damit keine Speicherlücken im Hauptspeicher entstehen. Wenn eine Schlange freigegeben wird, z.B. beim Beenden des Programms, müssen alle noch vorhandenen Schlangenelemente freigegeben werden. Wenn ein Schlangenelement freigegeben wird, sollte normalerweise auch das zugehörige Bezugsobjekt (hier der entsprechende Patient) freigegeben werden. Dies führt aber zu Problemen, wenn dieser Patient noch in einer anderen Schlange enthalten ist. Das Programm müsste sich also merken, ob Patienten in mehreren Schlangen enthalten sind. Dies ist oft die Ursache für schwerwiegende Programmierfehler, die in Java durch die automatisierte Garbage Collection entfallen.

Übung 4.4 Speichern Sie das Projekt als `Wartezimmer3`. Implementieren und testen Sie die Wartezimmersimulation wie in Abbildungen 4.5 und 4.8 angegeben. Das Programm soll genau wie das Programm aus Übung 4.3 funktionieren und die gleiche Benutzeroberfläche haben.

Übung 4.5 Speichern Sie das Projekt als `Wartezimmer4`. Erweitern Sie das Programm um eine zweite Patientenschlange, die auch Patienten enthält, von denen erst ein EKG gemacht werden muss, bevor sie ins Behandlungszimmer kommen. Die Programmoberfläche sehen Sie in Abbildung 4.9. Patienten mit EKG-Messung werden in beide Schlangen eingefügt. Die EKG-Schlange soll durch die Assistentinnen so rechtzeitig abgearbeitet werden, dass die Reihenfolge der Patienten für die Arztkonsultation entsprechend der Reihenfolge bei der Anmeldung erhalten bleibt. Bei der Anzeige der Schlangen soll die Zahl am Ende die Länge der Schlange angeben.

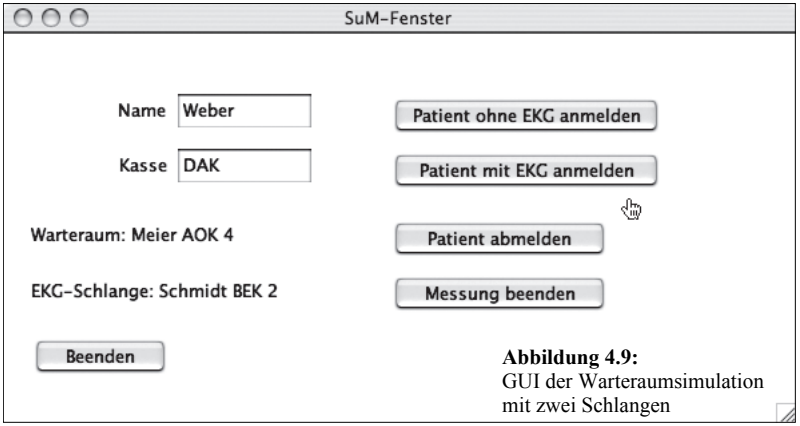


**Abbildung 4.9:**
GUI der Warteraumsimulation mit zwei Schlangen

## 4.4 Generische Schlangen

Bis jetzt wurden immer Patientenschlangen betrachtet, deren Elemente eine kennt-Beziehung zu Patienten besaßen. Da aber Schlangen bei vielen anderen Programmieraufgaben auftreten, ist es wünschenswert, so etwas wie eine allgemeine Schlange zu entwickeln, die Objekte anderer Art enthalten können. Am besten wäre es, wenn die Schlangenknoten sogar unterschiedliche Objektarten kennen könnten. Ein anschauliches Modell hierfür ist eine Wäscheleine, an der Hosen, Hemden, Socken usw. hängen können. Eine Schlange, die beliebige Objekte enthalten kann, bezeichnet man als *generische Schlange*.

Jede Java-Klasse ist automatisch eine Unterklasse der vordefinierten Klasse `Object`. Also enthält eine generische Schlange Objekte der Klasse `Object`.

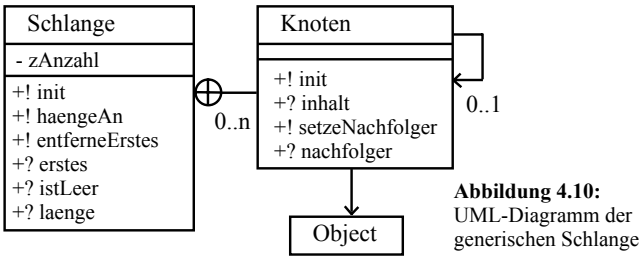


**Abbildung 4.10:**
UML-Diagramm der generischen Schlange

Im Quelltext der Patientenschlange muss also die Klasse `Patient` überall durch die Klasse `Object` ersetzt werden. Die Dienste `entferneErsten` und `erster` werden in `entferneErstes` und `erstes` umbenannt, da es sich jetzt nicht mehr um Patienten sondern um Objekte handelt.In der Klasse `Praxisanwendung` müssen zum Schluss noch kleine Änderungen durchgeführt werden.

Das Anhängen neuer Patienten kann unverändert bleiben. In der Klasse `Schlange` wurde das Anhängen neuer Objekte mit folgender Signatur definiert:

```
public void haengeAn(Object pInhalt)
```

Die Anweisung der Klasse `Praxisanwendung`

```
hatEKGSchlange.haengeAn(lPatient);
```

ist eine gültige Anweisung, da `lPatient` ein Objekt der Klasse `Patient` und damit auch der Oberklasse `Object` ist.

Der Zugriff auf das erste Element der Schlange ist etwas umständlicher. In der Klasse Schlange gibt es dazu folgenden Dienst:

```
public Object erstes()
```

Die Anfrage liefert also ein Objekt der Klasse `Object` zurück. Um diesen Rückgabewert dem Bezeichner `lPatient` zuweisen zu können, muss erst noch eine *Typkonvertierung* (engl. typecasting) durchgeführt werden. Dazu wird der Bezeichner der gewünschten Klasse in Klammer vor den Aufruf der Anfrage `erstes` geschrieben:

```
lPatient = (Patient)hatEKGSchlange.erstes();
```

Regel Beim Hinzufügen von Objekten in eine generische Schlange findet **keine** Typkonvertierung statt, da jede Klasse eine Unterklasse der Klasse `Object` ist. Beim Lesen eines Schlangenelements **muss eine Typkonvertierung stattfinden**, da die Klasse `Schlange` nur Objekte der allgemeinen Klasse `Object` zurückgibt. Die Typkonvertierung wird erreicht, indem der gewünschte Typ in Klammern vor die Anfrage an die Schlange geschrieben wird.

Übung 4.6 Speichern Sie das Projekt als `Wartezimmer5`. Implementieren Sie die Wartezimmersimulation mit der **generischen Schlange**. Das Programm soll sich dabei genau so verhalten, wie das Programm aus Übung 4.5.

## 4.5 Lokale Bibliotheken

In vielen Programmierprojekten werden Schlangen benötigt. Die generische Schlange aus dem vorigen Abschnitt ist bewusst allgemein gehalten, so dass sie in anderen Projekten benutzt werden kann. Java bietet dazu die Möglichkeit an, Klassen in einer Bibliothek zu speichern. Die Bibliothek enthält nur den übersetzten Java-Bytecode und nicht mehr den Quelltext der enthaltenen Klassen. Sie sollen jetzt lernen, wie man eine solche Bibliothek, die erst mal nur die Klasse Schlange enthalten soll, erzeugt und in BlueJ einbindet.

Zuerst müssen Sie sich einen Namen für die Bibliothek überlegen. Da die Bibliothek später noch andere Klassen zur Verwaltung von Datenstrukturen enthalten soll, empfiehlt sich das Wort strukturen. Außerdem ist es üblich, dass man an der Bibliotheksbezeichnung erkennen soll, woher die Bibliothek stammt. Hier soll könnte man die Kurzbezeichnung der Schule wählen, z.B. **bspg** für Beispiel-Gymnasium. Die Bibliothek soll also den Namen `bspg.strukturen` erhalten. Wenn Sie dann in einem anderen Projekt die Klasse `Schlange` benutzen wollen, schreiben Sie nur

```
import bspg.strukturen.*;
```

Wie erzeugt man jetzt eine solche Bibliothek?

• Speichern Sie das Wartezimmerprojekt unter einem neuen Namen z.B. `Strukturen1`
• Entfernen Sie alle Klassen außer der Klasse `Schlange`
• Wählen Sie im Menü `Bearbeiten - Neues Paket...`
• Nennen Sie das neue Paket `bspg.strukturen`
• Öffnen Sie die Klasse `Schlange`
• Ergänzen Sie als erste Zeile `package bspg.strukturen;`
• Schließen Sie das Fenster mit dem Programmtext und wählen Sie `Verschieben`

Jetzt befindet sich der Quelltext der Klasse im Ordner `strukturen`, der im Ordner `bspg` enthalten ist.

• Wählen Sie im Menü `Projekt - Als jar-Archiv speichern...`
• Ändern Sie nichts an den Einstellungen und wählen Sie `Weiter`
• Speichern Sie die Datei unter dem Namen `bspg.strukturen` (Die Endung `jar` wird automatisch angehängt.)

Damit haben Sie die Bibliothek erzeugt. Um die Bibliothek benutzen zu können gibt es mehrere Methoden.

• Sie legen `bspg.strukturen.jar` in den Ordner `extensions` der sich im BlueJ lib-Ordner befindet (siehe Kapitel 1). Dies ist aber in der Schule oft nur mit Administratorenrechten möglich.

• Sie öffnen die Einstellungen von BlueJ und können unter der Einstellung für Bibliotheken den Pfad zu einer Bibliothek (jar-Datei) angeben. Anschließend muss BlueJ neu gestartet werden, damit die Bibliothek geladen wird und anderen Klassen zur Verfügung steht. Ob die Bibliothek geladen wurde können Sie in den BlueJ-Einstellungen kontrollieren.

• Sie erzeugen einen neuen Ordner mit der Bezeichnung **`+libs`**, in den Sie die Datei `bspg.strukturen.jar` legen. Diesen neuen Ordner legen Sie in den entsprechenden aktuellen Projektordner z.B. `Wartezimmer5`. Jetzt steht diese Bibliothek dem Projekt zur Verfügung. Diese Methode der Einbindung einer Bibliothek soll im weiteren Verlauf immer wieder benutzt werden.

Zum Schluss entfernen Sie im Projekt die Klasse `Schlange` und ergänzen in der Klasse `Praxisanwendung` die Zeile

```
import bspg.strukturen.*;
```

Jetzt können Sie das Projekt neu übersetzen und ausführen. Die Klasse `Schlange` der Bibliothek wird jetzt benutzt. Auf die gleiche Art wurden auch die verschiedenen SuM-Bibliotheken erzeugt. Beachten Sie, dass Sie in diesem Abschnitt statt `bspg` eine Abkürzung für Ihre Schule verwenden sollten.

Java stellt standardmäßig eine umfangreiche Klassenbibliothek mit mehreren tausend Klassen zur Verfügung. In BlueJ können Sie im Hilfe-Menü `Java Klassenbibliothe-`

ken aufrufen und die Dokumentation der mitgelieferten Klassen ansehen. Auch hier sind die Bibliotheksklassen durch Pakete gegliedert. Sie erkennen hier auch, wie wichtig es ist, eine Klasse mit JavaDoc vernünftig zu dokumentieren, ansonsten ist eine Bibliothek für andere Benutzer ziemlich wertlos.

Übung 4.7 Speichern Sie das Projekt als `Wartezimmer6`. Implementieren Sie die Wartezimmersimulation mit der **bspg.strukturen-Bibliothek**.

## 4.6 Anzeigen der Schlange

Oft will man zu Testzwecken den Inhalt der Schlange ansehen können. Dazu benutzt man am besten einen Zeilenbereich. Bei der Wartezimmersimulation gibt es zwei Warteschlangen, die in zwei Zeilenbereichen dargestellt werden sollen. Eine mögliche Benutzeroberfläche sehen Sie in Abbildung 4.11.

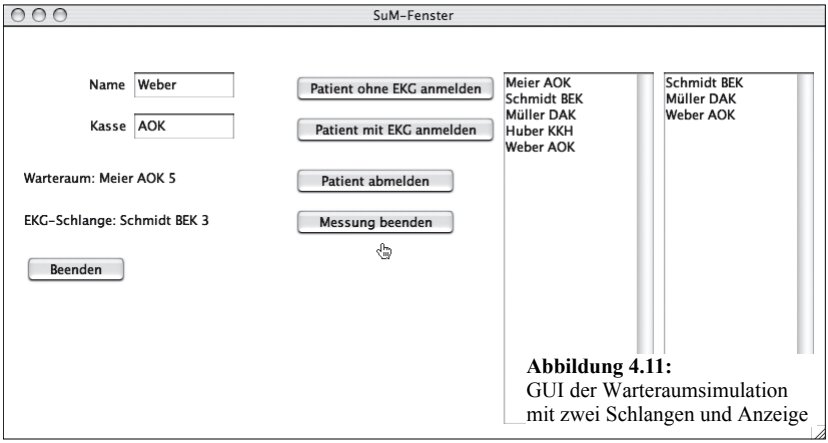


**Abbildung 4.11:** GUI der Warteraumsimulation mit zwei Schlangen und Anzeige

Die Klasse Schlange erhält einen Dienst `toString`, der eine Stringdarstellung der Schlange zurückgibt. Dazu wird die Schlange in einer Schleife durchlaufen und die jeweiligen Inhalte der Schlange werden an den String angehängt. Der Dienst `toString` ist schon in der Klasse `Object` definiert, die Oberklasse aller anderen Klassen ist. Er wird also in der Klasse `Schlange` überschrieben.

Im Zeilenbereich soll jede Zeile eines der in der Schlange gespeicherten Objekte als String darstellen. Glücklicherweise besitzt die Klasse `Object` eine Anfrage `toString`, die das Objekt als Stringrepräsentation zurückgibt. Dieser Dienst muss in den Klassen, deren Objekte in die Schlange kommen, ergänzt und damit überschrieben werden. In der Klasse `Patient` muss also auch die Anfrage `toString` ergänzt werden:

```
public String toString()
{
   return zName + " " + zKasse;
}
```

Die Anfrage `toString` der Klasse `Schlange` lautet dann folgendermaßen:

```
/**
 * Eine Stringdarstellung der Schlange wird zurückgegeben.
 * @return die Stringdarstellung der Schlange
 */
public String toString()
{
  String lString;
  Knoten lKnoten;

  lString = "";
  lKnoten = kenntKopf;
  if (this.istLeer())
    lString = "leere Schlange";
  else
    while (lKnoten != null)
    {
      lString += lKnoten.inhalt().toString() + "\n";
      lKnoten = lKnoten.nachfolger();
    }
  if (hatTW.laenge(lString) > 0)
  lString = hatTW.textOhne(lString, hatTW.laenge(lString),
                           hatTW.laenge(lString));
    return lString;
}
```

In der Schleife wird am Ende das Zeichen \n angehängt, dadurch wird ein Zeilenvorschub erzeugt. So wird jedes Schlangenelement in einer eigenen Zeile dargestellt. Mit einem Textwerkzeug wird zum Schluss der letzte Zeilenvorschub entfernt.

In der Praxisanwendung muss nach jeder Anmeldung und Abmeldung die entsprechende Schlangenanzeige aktualisiert werden.

```
hatWartebereich.setzeInhalt(hatPatientenschlange.toString());
hatEKGBereich.setzeInhalt(hatEKGSchlange.toString());
```

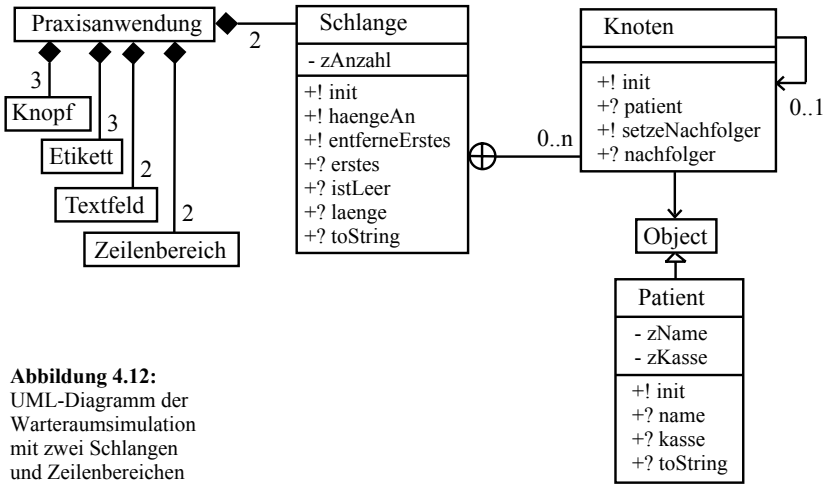


**Abbildung 4.12:**
UML-Diagramm der Warteraumsimulation mit zwei Schlangen und Zeilenbereichen

Übung 4.8 Öffnen Sie das Projekt `Strukturen1` und speichern Sie es als `Strukturen2`. Ergänzen Sie den Dienst `toString` in der Klasse `Schlange`. Erzeugen Sie die jar-Datei `bspg.strukturen.jar`. Speichern Sie das Projekt `Wartezimmer6` als `Wartezimmer7`. Ergänzen Sie die lokale Bibliothek mit der neuen jar-Datei. Implementieren Sie die Wartezimmersimulation mit Anzeige der beiden Warteschlangen.

Übung 4.9 Schreiben Sie ein Programm zur Simulation einer Tanzschule. In der Tanzschule treffen neue Personen ein, die entweder in die Schlange der Damen oder die Schlange der Herren eingereiht werden. Anschließend werden Paare gebildet, die in eine dritte Schlange, die Tanzschlange, eingereiht werden. Paare verlassen die Tanzschlange in der Reihenfolge ihrer Ankunft. Benutzen Sie dabei die Klassen `Dame`, `Herr` und `Paar`.

## 4.7 Zusammenfassung

In diesem Kapitel haben Sie als erste lineare Datenstruktur die Klasse `Schlange` kennen gelernt. Ausgehend von einer einfachen Patientenverkettung wurde das Konzept der Schlange immer weiter verallgemeinert, bis Sie zum Schluss die generische Schlange, die beliebige Objekte enthält, implementiert haben. Datenstrukturen, die mehrere Objekte verwalten, werden auch als *Sammlungen* (engl. collections) bezeichnet.

Das Besondere an einer Schlange ist die Eigenschaft, dass man neue Objekte nur **am Ende** der Schlange **anfügen** kann, aber Objekte nur **vom Anfang** (Kopf) der Schlange **entfernen** kann. Dies bezeichnet man als **FIFO**-Prinzip (FIFO = First In - First Out). Die Elemente werden in der Reihenfolge, in der sie angehängt wurden, wieder aus der Schlange entfernt.

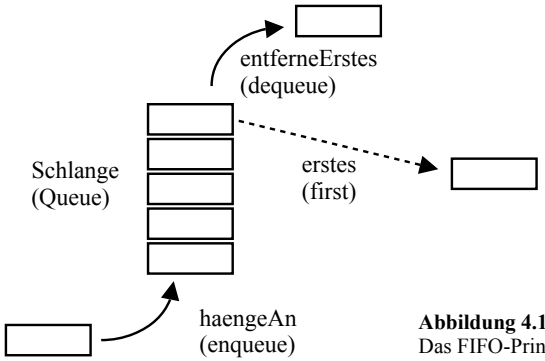


**Abbildung 4.13:**
Das FIFO-Prinzip

Schlangen spielen in der Informatik (wie im täglichen Leben) eine bedeutende Rolle. Betriebssysteme verwalten Warteschlangen für Druckaufträge; die Zeichen, die auf einer Computertastatur getippt werden, bilden eine Schlange; Datenpakete, die im Internet übertragen werden, bilden Schlangen, die von Routern verwaltet werden.

Schlangen werden im Englischen als *Queue* bezeichnet und die Standarddienste mit *enqueue* für das Anhängen und *dequeue* für das Entfernen des ersten Elements bezeichnet.

In Abbildung 4.14 sind die entsprechenden Klassendiagramme gegenübergestellt.

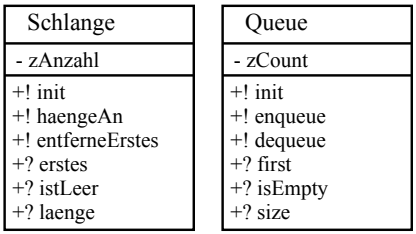


**Abbildung 4.14:**
Schlange (deutsch)
Queue (englisch)

Zum Schluss haben Sie gelernt, wie man eine eigene Klassenbibliothek (engl. library) erzeugt und in ein Programm bzw. Projekt einbindet. Diese Bibliothek soll in den nächsten Kapiteln durch weitere Klassen ergänzt werden. Das Besondere an einer Bibliothek ist, dass die enthaltenen Klassen nicht mehr im Quelltext sondern nur in übersetzter Form zur Verfügung stehen. Man kann die Klassen also nicht mehr direkt verändern sondern muss eventuell Unterklassen bilden.

In BlueJ gibt es drei Möglichkeiten Bibliotheken in ein Projekt einzubinden:

(1) Man erzeugt im Projektordner einen Ordner mit dem Namen `+libs`, in den man die jar-Datei legt. Die Bibliothek steht jetzt genau diesem Projekt zur Verfügung.

(2) Man wählt in BlueJ den Menüpunkt `Einstellungen`. Unter dem Punkt `Bibliotheken` kann man Bibliotheken (also die jar-Dateien) hinzufügen bzw. löschen. Allerdings muss BlueJ anschließend neu gestartet werden, damit die Bibliothek geladen wird. Die Bibliothek steht jetzt allen BlueJ-Projekten zur Verfügung.

(3) Man legt die jar-Datei im BlueJ-Programmordner im Unterordner `lib` in den Unterordner `userlib`. Auch jetzt wird die Bibliothek beim nächsten Start von BlueJ geladen und steht allen BlueJ-Projekten zur Verfügung. Dieses Verfahren wurde im Kapitel 1 Abschnitt 1.3 benutzt, um die SuM-Bibliotheken zu ergänzen.

Da in den nächsten Kapiteln die bspg.strukturen-Bibliothek immer wieder verändert werden wird, empfiehlt sich das erste Verfahren (1), also die Benutzung lokaler Bibliotheken.

BlueJ bietet die Möglichkeit, ein Objekt der Klasse `Schlange` auf der Werkbank zu erzeugen. Wählen Sie dazu im Menü `Werkzeuge - Klasse aus Bibliothek verwenden...` und geben Sie `bspg.strukturen.Schlange` ein. Nachdem Sie `Return` gedrückt haben, können Sie den Konstruktor der Schlange doppelklicken und ein Objekt der Klasse `Schlange` wird auf der Werkbank erzeugt. Jetzt können Sie die Funktionalität der Klasse interaktiv testen.

## Neue Begriffe in diesem Kapitel

• **Schlange** (engl. queue) Eine Schlange ist eine lineare Struktur (Sammlung), in der Objekte nach dem FIFO-Prinzip verwaltet werden. Wenn eine Schlange beliebige Objekte verwalten kann, spricht man von einer *generischen Schlange*.

• **FIFO** steht für *First In First Out*, das Prinzip, wie Elemente einer Schlange verwaltet werden. Die Elemente verlassen die Schlange in der Reihenfolge ihrer Ankunft.

• **Klassenbibliothek** jar-Datei (jar = Java Archiv), die übersetzte Klassen (ohne Quellcode) enthält, die anderen Klassen zur Verfügung gestellt werden können.

## Java-Bezeichner

• **package** (deutsch Paket) Jede Klasse einer Java-Bibliothek gehört zu einem Paket. Als erste Zeile im Quelltext der Klasse steht die Paketinformation, z.B. `package bspg.strukturen;`. Sämtliche Klassen eines Pakets sind in einem gemeinsamen Projekt implementiert. Aus diesem Projekt kan man eine jar-Datei erzeugen, die den übersetzten Quelltext in komprimierter Form enthält.

• **jar** (Abkürzung für Javaarchiv) Endung einer Datei, die ein Paket von übersetzten Java-Klassen enthält. Eine jar-Datei ist entweder ein Paket, dessen Klassen von einem Projekt benutzt werden können oder ein ausführbares Java-Programm, das mit einem Doppelklick gestartet werden kann.