Objektorientierte Softwareentwicklung

Sven Eric Panitz
Hochschule RheinMain

Version 1423

Generiert von LectureNotes Teaching System
4. Oktober 2019

Dieses Skript hat eine lange Tradition. Es basiert in Teilen auf den Skripten, die seit 2002 für unterschiedliche Vorlesungen im Bereich der Softwareentwicklung entstanden sind. Seit 2002 hat sich viel in Java getan: es kamen Generics, Aufzählungstypen und For-Each-Schleifen in Java 5, Lambdas und Streams in Java 8, Typinferenz für lokale Variablen und die jshell in Java 10 und aktuell ab Java12 geht es um die Ausweitung der switch-Anweisung hin zu einem vollen Pattern-Matching.

Ich bemühe mich stets auf diese neuen Eigenschaften einzugehen und ein aktuelles Begleitskript zu meiner Vorlesung vorzulegen. Es ist zusammen mit den Aufgaben und Lernkarteikarten auf meiner Lernplattform `subato.org` zu verwenden.

Hatte zwischenzeitig mein Skript über 500 Seiten und sollte möglichst viel Details in Java abbilden, habe mich entschieden es wieder auf wesentliche Inhalte einzudampfen. Ein komplettes Java-Kompendium und Nachschlagwerk findet man in Standardbüchern. Dieses Skrpit ist kein Buch sondern begleitet die Vorlesung zu meinen Modul. Es ist auch nicht mit der gleichen Sorgfalt wie ein Buch geschrieben und enthält. Eine Reihe von wichtigen Java-Themen werden auch erst im getrennten Skript des Moduls »Programmiermethoden und Techniken« des zweiten Semesters behandelt,

Inhaltsverzeichnis

1 Einführung in die Welt der Softwareentwicklung

Sie haben sich für das Studium eines Informatikstudienganges entschieden. Sie werden damit u.a. in den verschiedenen Modulen verschiedene Aspekte zur Entwicklung von Software kennenlernen. Dieses wird eines der zentralen Inhalte des Studiums sein. Und in diesem Modul werden die Grundlagen hierfür gelegt.

1.1 Aspekte der Softwareentwicklung

> Ich schreibe. Ich bin Software-Autor. [...] Ich schreibe ab jetzt nur noch Software, die was mit meinem Leben zu tun hat. Autobiographisch.
>
> *Kristof Magnusson: Männerhort*

Das obige Zitat aus der Komödie »Männerhort« ist näher an der Realität, als man denken mag; denn im Prinzip ist Software nichts weiter als ein Text, den ein oder mehrere Autoren geschrieben haben. Insofern ist es durchaus legitim bei Programmierern von Autoren zu sprechen. Und je mehr sich die Autoren mit ihrer Software identifizieren, desto besser ist meistens auch die Qualität der Software. Dieses gilt oftmals besonders für quelltextoffene Software. Hier steht immerhin der gute Ruf der Autoren auf dem Spiel.

Auf der einen Seite scheint die Entwicklung von Software einfach zu sein. Es ist ein Text zu schreiben. Hierzu wird eigentlich kein anderes Werkzeug als ein Texteditor benötigt. Andererseits ist die Entwicklung von Software ein sehr komplexer Vorgang, denn ein Softwaresystem ist oft sehr umfangreich und besteht aus vielen Einzelkomponenten. Um dieser Komplexität Herr zu werden, hat die Informatik über die letzten Jahrzehnte viele Techniken und Prozesse entworfen, die bei der Entwicklung von Software helfen sollen. Im Laufe eines Informatikstudiums werden Sie in verschiedenen Modulen die vielen unterschiedlichen Aspekte der Software-Entwickung kennenlernen. In diesem Modul soll Ihnen hierzu ein Grundüberblick und ein erstes Fundament in der Programmierung gelegt werden.

Die Fragestellungen der Softwareentwicklung reichen von technischen Aspekten, wie die Programmierung von Betriebsmitteln, bis hin zu Prozessen und Rollen beim Arbeiten im Team. Es ist zu klären, für welche Hardware was für Software mit welcher Funktionalität

auf welche Weise zu entwickeln ist, wie die Qualität der Software sicher gestellt werden kann, wie die Kosten des Entwicklungsprozesses abgeschätzt werden können und wie die Software zum Kunden gelangt und im laufenden Betrieb gewartet werden kann.

Wir geben eine ungeordnete und sicher nicht vollständige Liste verschiedenster Aspekten der Softwareentwicklung:

- **Anforderungsanalyse**: Am Anfang steht die banale Frage, für wen Software erstellt werden soll und was diese Software ganz allgemeoin können muss. Was ist der Einsatzbereich. Wer sind die Anwender. Die Analyse dieser solcher Fragen wird Anforderungsanalyse genannt. Allein aus der Tatsache, dass wir im Wahlpflichbereich ein eigenes Modul zu diesem Thema zeigt, dass die Anforderungen oft nicht so offensichtlich sind, wie es scheint.
- **Spezifikation**: Für einzelne Komponente und Funktionalitäten ist zu klären, was diese für ein genaues Eingabe- Ausgabeverhalten haben sollen. Diesen Schritt der Softwareentwicklung bezeichnet man allgemein als Spezifikation. Spezifiziert wird auf den unterschiedlichsten Abstraktionsebenen. Allgemein für die komplette Software, für einzelne Komponenten oder auch für kleinste Teileinheiten. Es gibt verschiedene formale und semi-formale Sprachen, zur Spezifikation. Im Bachelorstudiengang werden Sie die gängigsten Spezifikationssprachen in dem Modul »Softwaretechnik« kennen lernen. Weitere formale Spezifikationssprachen werden im Masterstudiengang im Modul »Formale Modelle« vorgestellt. Grundlage aller fast Spezifikationssprachen ist ein gewisses formales Handwerkszeug, dass im Modul »Diskrete Strukturen« vermittelt wird.
- **Sicherheit und Datenschutz**: Wer die Tagespresse verfolgt, sieht, dass die Sicherheit von Daten ein großes und ernstes Thema ist. Das Modul »IT-Security« wird hierzu in Ihren Studium die theoretischen Grundlagen dieses Aspektes vermitteln.
- **Ergonomie und Benutzerfreundlichkeit** Jeder Anwender hat sich sicher schon einmal darüber geärgert, dass Software nicht intuitiv oder folgerichtig zu bedienen war. Das einfache Dinge nur kompliziert und durch viele Schritte realisiert werden konnten oder auch, dass die Software die wichtigen Informationen nicht leicht zugänglich und übersichtlich dargestellt hat. Auch eine in der Funktionalität fehlerfreie Software kann unbrauchbar sein, weil sie zu schwer zu bedienen ist. Im Wahlbereich unseres Curriculums findet sich das Modul »Usability Engineering«, dass sich mit diesen Fragen auseinander setzt.
- **Optimierung** Ein wichtiger Aspekt einer Software ist, dass sie nicht nur korrekt funktioniert, sondern auch einen guten Durchsatz und schnelle Antwortzeiten hat. Hierbei kann eine schnelle Antwortzeit aber auch ganz Unterschiedliches bedeuten. Von einer Handelsplattform für Börsengeschäfte wird eine extrem schnelle Abarbeitung der Anfrage erwartet, ein System das ein dreidimensionalen Film rendered darf durchaus manchmal Stunden brauchen. Egal in welchem Gebiet die Software eingesetzt wird, sie soll möglichst effizient laufen. Hierbei wird es unter Umständen nötig, korrekt laufende Programme zu optimieren. Optimierungen können den verwendeten Algorithmus betreffen, der in einer hohen Komplexitätsklasse liegt und

durch einen effizienteren ersetzt werden kann. Es kann das Zwischenspeichern von Teilergebnissen beinhalten, so dass mehrfaches Errechnen gleicher Zwischenergebnisse entfällt. Es kann aber auch auf einer mikroskopischen Ebene passieren, indem besonders häufig benutze Programmteile schon in kleinsten Teilen auf Assemblerebene optimiert werden. Mit der Komplexität bestimmter Algorithmen beschäftigt sich in unserem Curriculum unter anderem das Modul »Algorithmen und Datenstrukturen«.

- **Tests** Beim Theater sagt man: »Alles, was nicht geprobt ist, geht schief.« Ähnliches gilt für Software. Alles, was nicht getestet wurde, hat eine hohe Wahrscheinlichkeit, dass es nicht funktioniert. Doch wie testet man Software am besten? Reicht es ein paar Anwender das Programm ausprobieren zu lassen? Welche Komponenten lassen sich automatisch reproduzierbar testen? Welche Granularität können die Tests haben? Wird in Kenntnis des Quellcodes getestet oder ohne diese Kenntnis das Ein-/Ausgabeverhalten der Software? Und wer testet? Die Entwickler selbst, oder Personen, die die Software nicht entwickelt habe? In diesem Modul werden wir erste Testszenarien für Entwicklertests kennenlernen. Ein umfassender Überblick wird in dem Modul »Softwaretechnik« gegeben.

- **Logging** Besonders für Server-Anwendungen, die über einen langen Zeitraum laufen, ist es wichtig, dass ein Administrator nachverfolgen kann, was über einen bestimmten Zeitraum mit der Anwendung vorgefallen ist. Wie viele Anfragen bedient wurden, ob es Fehlerfälle gab oder auch wie gut der Durchsatz der Anwendung war. Hierzu ist es unerlässlich, dass eine Anwendung ein Logbuch führt, in dem alle wichtigen Schritte des Prorammablaufs dokumentiert sind. Dieses Logbuch ist insbesondere wichtig, wenn es zu Fehlern kam, und die Fehlerursache ermittelt werden muss.

- **Verifikation** Klar, Software soll fehlerfrei laufen. Aber nicht immer genügen Tests, um das fehlerfreie Verhalten eines Programms zu zeigen. Tests können immer nur endlich viele Eingaben eines Programms checken. Ein Programm hat aber potentiell unendlich viele Zustände und Eingaben. Mit formalen Verifikationsmethoden versucht man für besonders kritische Software, die vielleicht eine gefährliche Anlage steuert, mathematisch zu beweisen, dass sie für alle Eingaben immer das gewünschte Verhalten zeigt. Man unterscheidet:

  - *partielle Korrektheit:* wenn das Programm für eine bestimmte Eingabe ein Ergebnis liefert, dann ist dieses bezüglich der Spezifikation korrekt.

  - *totale Korrektheit:* Das Programm ist partiell korrekt und terminiert für jede Eingabe, d.h. liefert immer nach endlich langer Zeit ein Ergebnis.

- **Internationalisierung** Die Welt ist insbesondere durch die Daten-Kommunikationsnetze enger zusammen gerückt. Ein Stück Software wird nicht für ein Land oder einen Kulturkreis allein geschrieben, sondern wird potentiell weltweit eingesetzt. Unter der Internationalisierung (abgekürzt als I18N für internationalization, der Buchstabe I 18 weitere Buchstaben und schließlich der Buchstabe N) versteht

man, dass die Software in verschiedenen Kulturkreises eingesetzt werden kann. Dieses beinhaltet z.B. dass alle Schriftzeichen in einer Textverarbeitung benutzt werden können oder dass die Schreibrichtung in einem Texteingabefeld eingestellt werden kann, da in einigen Schriften von rechts nach links in anderen von links nach rechts, und teilweise auch von oben nach unten geschrieben wird.

- **Lokalisierung** Die Lokalisierung beschäftigt sich auch mit der globalen Einsetzbarkeit der Software. Hier geht es zusätzlich darum, dass die Benutzerführung in verschiedenen Sprachen vorliegt, zum Beispiel die Menübeschriftungen in mehreren Sprachen vorliegen und die Hilfetexte übersetzt worden sind. Entsprechend I18N wird Lokalisierung auch als L12N abgekürzt. Kurz gefasst kann man sagen: in einer internationalisierten Software lassen sich auch chinesische Texte schreiben, aber die Knöpfe und Menüs sind weiterhin auf englisch beschriftet. In einer lokalisierten Software gibt es auch Beschriftungen, Hilfetexte etc auf chinesisch.
- **Vorgehensmodell** Wie schreibt man eine Software. Einfach los tippen, bis man fertig ist, wäre ein Vorgehensmodell. Ein Vorgehensmodell legt fest, in welchen Schritten, mit welchen Zwischenergebnissen eine Software im Team erstellt wird. Dabei kann das Team auch aus einer Person bestehen, wie es in der Regel bei einer Bachelorarbeit der Fall ist. Es wurden in der Informatik eine Reihe unterschiedlicher Vorgehensmodelle entwickelt. Auf theoretischer Ebene werden diese Ihnen im Modul »Softwaretechnik« vorgestellt. Praktisch werden Sie spätestens im Wahlprojekt des fünften Semesters nach einem Vorgehensmodell arbeiten.
- **Dokumentation** Schön wäre es, wenn der Quelltext einer Software selbsterklärend wäre. manche Programmierer sind auch der Ansich: »The code is documentation enough.«. Dieses ist leider selten der Fall, auch wenn es auch eine Bestrebung in der Entwicklung neuer Programmiersprachen ist, dass Programme verständlicher und selbsterklärender sind. Daher ist es unumgänglich, dass zu einer Software eine Dokumentation existiert. Wobei man drei Adressaten der Dokumentation identifizieren kann:
  - **Endanwender**: für die Endanwender wird ein Benutzerhandbuch geschrieben. Hier sind keinerlei Interna der Software beschrieben, sondern lediglich die Funktionalität der Benutzerschnittstelle.
  - **Entwickler die Komponenten benutzen**: die meiste Software ist nicht monolithisch, sondern besteht aus einzelnen Komponenten, die in unterschiedlichen Softwareprojekten wiederverwendet werden können. Für diese Komponenten ist die Programmierschnittstelle, das sogenannte API (*application programmers interface*) zu beschreiben. Es ist zu dokumentieren, welche Funktionalität die Komponente anbietet und wie diese von einem Programmierer zu benutzen ist. Welche Vorbedingungen gelten müssen und welche Nachbedingungen nach Aufruf einer Schnittstelle gelten. Interna der Umsetzung sind hierbei nicht dokumentiert.
  - **Entwickler der Software selbst:** Sie werden schon in Kürze feststellen, dass Sie Ihre eigenen Programme schon nach wenigen Wochen manchmal gar Tagen

nicht mehr verstehen. Sie werden sich fragen: was habe ich mir da nur gedacht? Deshalb ist es sinnvoll, seinen Programmtext zu kommentieren. Innerhalb des Programmtextes sind Kommentarzeilen eingefügt, die erklären, wie ein bestimmter Teil des Codes funktioniert. Diese Kommentare sollen helfen, dass die Kollegen im Team oder man selber, oder Kollegen, die irgendwann in der Zukunft die Software weiterentwickeln, den Programmtext besser verstehen.

In vielen Firmen verlangt wird das API auf Englisch zu dokumentieren und auch Kommentare im Programmtext auf Englisch zu schreiben. Dieses soll erleichtern, dass auch internationale Kollegen mit ins Team aufgenommen werden können oder die Software ins Ausland zur Weiterentwicklung verkauft werden kann. Auch bei einer kleinen Firma, die in Deutschland angesiedelt ist und nur deutsche Mitarbeiter hat, weiß man nie, was die Zukunft bringt. Vielleicht wird die Firma von einer anderen in Korea angesiedelten Firma aufgekauft, vielleicht wird es eine lose Zusammenarbeit mit einer französischen Firma geben. Auf jeden Fall sind solche Schritte weniger verbaut, wenn der Programmtext auf Englisch dokumentiert und kommentiert ist. Es empfiehlt sich also, sich frühzeitig anzugewöhnen, seinen Quelltext in seinem rudimentären Englisch zu dokumentieren.

- **Wartung** Wenn Sie in Ihrem betriebspraktischen Modul in einer Firma sind, die Software entwickelt, werden sie vielleicht feststellen, dass dort mehr Zeit damit verbracht wird, bestehende Software zu warten als neue Software zu schreiben. Zur Wartung gehört zunächst einmal, dass Fehler, die erst im Betrieb beim Kunden gefunden wurden, korrigiert werden. Es kann sich dabei um echte funktionale Fehlfunktionen, die bis zum Programmabsturz führen können, handeln aber auch um Probleme der Benutzerfreundlichkeit und der Ausführungsgeschwindigkeit. Einen großen Wartungsaufwand fordern oft neue Versionen des Systems, auf dem die Software installiert ist. Sei es das Betriebssystem, ein Webserver oder auch die Datenbank.
- **Portieren:** Oft wird es nötig, ein Programm auf eine andere Plattform zu portieren. Ein unter Windows erstelltes Programm soll z.B. auch auf Unix-Systemen zur Verfügung stehen.
- **Einsatz eines Debuggers** Bei einem Programm ist immer damit zu rechnen, dass es Fehler enthält. Diese Fehler werden im besten Fall von der Qualitätssicherung entdeckt, im schlechteren Fall treten sie beim Kunden auf. Um Fehler im Programmtext zu finden, gibt es Werkzeuge, die ein schrittweises Ausführen des Programms ermöglichen *(debugger)*. Dabei lassen sich die Werte, die in bestimmten Speicherzellen stehen, auslesen und auf diese Weise der Fehler finden.
- **Architektur** Eine Software besteht aus vielen Einzelkomponenten, die unterschiedliche Teilaufgaben übernehmen und im besten Falle für verschiedene Softwareprojekte wiederverwendbar sind. Die Aufteilung der Software in einzelnen Komponenten wird als Architektur bezeichnet. Es gibt bestimmte Standardarchitekturmuster, die bei ähnlichen Aufgabenstellungen Anwendung finden können. Elementare Programmiermuster werden im Modul »Softwaretechnik« vorgestellt.

- **Kommunikation** Es gibt heute kaum noch Software, die nicht auf irgendeine Art und Weise über ein Netzwerk kommuniziert. Insofern ist ein wichtiger Aspekt der Softwareentwicklung, wie diese Kommunikation gemacht wird. Insbesondere auch wie eine sichere Kommunikation erreicht wird und wie sich die Software bei Ausfall der Kommunikationswege verhalten soll. Die Grundlagen zur Datenkommunikation werden im Modul »Rechnernetze und Telekommunikation« vermittelt. Mit Anwendungen, die stark auf dem Internet basieren, beschäftigt sich das Modul »Webbasierte Anwendungen«.

- **Datenhaltung** Kaum ein Softwareprojekt kommt ohne eine Komponente zur Speicherung umfangreicher Daten aus. Man spricht dabei von Persistenz. Daten bleiben erhalten, auch wenn das Programm nicht mehr läuft. Eine einfache Form der Datenpersistenz ist eine Datei, in der die Daten in irgendeiner Form geschrieben werden. In der Regel haben Daten eine Struktur und werden in einer strukturierten Form abgespeichert, um in dieser Struktur effizient suchen zu können. Hierzu gibt Datenbanken. Bereits in zweitem Semester werden Sie im entsprechendem Modul die Grundlagen der Datenbanktechnologie zur Softwareentwicklung kennen lernen.

- **Algorithmen** In der Regel erhalten Unterprogramme bestimmte Eingaben, für die eine Ausgabe zu berechnen ist. Der Weg aus den Einzelschritten, wie diese Ausgabe erzeugt wird, wird als Algorithmus bezeichnet. Für viele typische immer wieder benötigte Aufgaben, wie zum Beispiel dem Sortieren einer Liste von Daten, gibt es Standardalgorithmen, auf die in der Software-Entwicklung zurück gegriffen werden kann. Im zweiten Semester im Modul »Algorithmen und Datenstrukturen« werden die elementarsten Standardalgorithmen vermittelt, sowie grundlegende Gedanken über die Komplexität bestimmter Algorithmen.

- **Codierung** Die eigentliche Codierung ist in der Regel der einzige Schritt, der direkt Code in der gewünschten Programmiersprache von Hand erzeugt. Alle anderen Schritte der Programmierung sind mehr oder weniger unabhängig von der zugrunde liegenden Programmiersprache.

  Für die Codierung empfiehlt es sich, Konventionen zu verabreden, wie der Code geschrieben wird, was für Bezeichner benutzt werden, in welcher Weise der Programmtext eingerückt wird. Entwicklungsabteilungen haben zumeist schriftlich verbindlich festgeschriebene Richtlinien für den Programmierstil. Dieses erleichtert, den Code der Kollegen im Projekt schnell zu verstehen.

- **Refaktorisierung** Man sagt zwar gerne »Never change a running system.« oder auch »If it works, don not fix it«, doch können wir das in der Software-Entwicklung so nicht stehen lassen. Funktionierende Lösungen können unnötig komplex und unübersichtlich sein, so dass sie auf lange Sicht schwer zu warten sind und auch nicht leicht um weitere Funktionalität erweitert werden kann. Hier ist es oft sinnvoll, den Code zu refaktorieren, d.h. umzuschreiben, so dass er durch eine einfachere und übersichtlichere Lösung ersetzt werden kann. Dieses kann man nur sicher machen, wenn es eine ausreichende Testabdeckung gibt, so dass man zeigen kann, dass die neue Lösung sich ebenso wie die alte verhält. Oft ist die Refaktorierung das Resultat

eines Reviews zum Beispiel in Form eines *code walk-through* bei dem der Entwickler seinen Programmtext zeilenweise zusammen mit anderen Entwicklern durchgeht.

Wie Sie sehen, gibt es einiges zu lernen in Ihrem Studium. Auf jeden Fall wird Ihnen nicht langweilig werden. Beginnen wir also mit den ersten Schritten zur Ausbildung zum Software-Autor.

1.2 Programmiersprachen

Auch wenn der letzte Abschnitt gezeigt hat, dass die eigentliche Codierung der Software nur ein kleiner Teil des Projektes ausmacht, sind doch eine oder mehrere Programmiersprachen auszuwählen, in denen das Projekt realisiert wird.

Es gibt mittlerweile mehr Programmiersprachen als natürliche Sprachen.[1] Die meisten Sprachen führen entsprechend nur ein Schattendasein und die Mehrzahl der Programme konzentriert sich auf einige wenige Sprachen. Programmiersprachen lassen sich nach den unterschiedlichsten Kriterien klassifizieren.

Im folgenden eine hilfreiche Klassifizierung in fünf verschiedene Hauptklassen.

- **imperativ** (C, Pascal, Fortran, Cobol): das Hauptkonstrukt dieser Sprachen sind Befehle, die den Speicher manipulieren.
- **objektorientiert** (Java, C++, C#, Objective C, Eiffel, Smalltalk): Daten werden in Form von Objekten organisiert. Diese Objekte bündeln mit den Daten auch die auf diesen Daten anwendbaren Methoden.
- **funktional** (Scala, Lisp, ML, Haskell, Scheme, Erlang, Clean, F#): Programme werden als mathematische Funktionen verstanden und auch Funktionen können Daten sein. Dieses Programmierparadigma versucht, sich möglichst weit von der Architektur des Computers zu lösen. Veränderbare Speicherzellen gibt es in rein funktionalen Sprachen nicht und erst recht keine Zuweisungsbefehle.
- **Skriptsprachen** (Javascript, Perl, AWK): solche Sprachen sind dazu entworfen, einfache kleine Programme schnell zu erzeugen. Sie haben meist kein statisches Typsystem und nur eine begrenzte Zahl an Strukturierungsmöglichkeiten, oft aber eine mächtige Bibliothek, um Zeichenketten zu manipulieren oder sehr einfache und intuitive Anweisungen, die es ermöglichen schnell eine lauffähige Lösung zu programmieren.
- **logisch** (Prolog): aus der KI (künstlichen Intelligenz) stammen logische Programmiersprachen. Hier wird ein Programm als logische Formel, für die ein Beweis gesucht wird, verstanden.

Viele moderne Programmiersprachen versuchen die besten Konzepte aus all diesen Welten zu kombinieren. Ein typischer Vertreter für eine Sprache, die das funktionale und das objektorientierte Paradigma vereint ist Scala.

[1] Wer Interesse hat, kann im Netz einmal suchen, ob er eine Liste von Programmiersprachen findet.

Eine weitere Unterscheidung von Programmiersprachen kann in der Art des Ausführungsmodells getroffen werden. Der Programmierer schreibt den lesbaren Quelltext seines Programms. Um ein Programm auf einem Computer laufen zu lassen, muss es erst in einen Programmcode übersetzt werden, den der Computer versteht. Für diesen Schritt gibt es auch unterschiedliche Modelle:

- **kompiliert** (C, C++, Cobol, Fortran): in einem Übersetzungsschritt wird aus dem Quelltext direkt das ausführbare Programm erzeugt, das dann unabhängig von irgendwelchen Hilfen der Programmiersprache ausgeführt werden kann.
- **interpretiert** (Lisp, Scheme, Javascript): der Programmtext wird nicht in eine ausführbare Datei übersetzt, sondern durch einen Interpreter Stück für Stück anhand des Quelltextes ausgeführt. Hierzu muss stets der Interpreter zur Verfügung stehen, um das Programm auszuführen. Interpretierte Programme sind langsamer in der Ausführung als übersetzte Programme.
- **abstrakte Maschine über *byte code*** (Java, Scala, Kotlin, ML): dieses ist quasi eine Mischform aus den obigen zwei Ausführungsmodellen. Der Quelltext wird übersetzt in Befehle nicht für einen konkreten Computer, sondern für eine abstrakte Maschine. Für diese abstrakte Maschine steht dann ein Interpreter zur Verfügung. Der Vorteil ist, dass durch die zusätzliche Abstraktionsebene der Übersetzer unabhängig von einer konkreten Maschine Code erzeugen kann und das Programm auf auf allen Systemen laufen kann, für die es einen Interpreter der abstrakten Maschine gibt.

Wir benützen in diesem Modul die Programmiersprache Java aus mehreren pragmatischen Gründen:

- Java ist objektorientiert und das objektorientierte Paradigma soll in diesem Modul vermittelt werden (dieses spricht zum Beispiel gegen Sprachen wie C, Pascal, Haskell).
- Java ist im Vergleich zu manch anderer Sprache relativ aufgeräumt und die Anzahl der Konzepte noch recht übersichtlich, so dass Java didaktisch zum Unterrichten gut geeignet ist (dieses spricht zum Beispiel gegen Sprachen wie C++ oder Scala).
- Java wird in der Praxis viel eingesetzt und ist für alle gängigen Plattformen geeignet. Dabei findet sich heutzutage Java nicht nur für Desktop und Server-Anwendungen sowohl auf Windows Betriebssystemen als auf Linux-Systemen, sondern auch für die Programmierung von Android Smartphones oder Webanwendungen mit Google Web Tool (dieses spricht zum Beispiel gegen Sprachen wie Objective C oder C#.)

Man kann trefflich über die Vor- und Nachteile unterschiedlicherer Sprachen streiten, sollte aber bei der Wahl der Programmiersprache stets pragmatisch sein und sich nicht zu sehr auf emotionale Diskussionen einlassen. Streits über unterschiedliche Programmiersprachen sind müßig. Jeder hat aus unterschiedlichen Gründen Favoriten, aber ein guter

Informatiker sollte sich nach etwas Einarbeitung in jeder Sprache zurecht finden. Insbesondere sollte man sich bei der Wahl der Programmiersprache von dem Einsatzgebiet der Software leiten lassen.[2]

1.3 Arbeiten mit der Kommandozeile

Wahrscheinlich ist das Hauptunterscheidungsmerkmal zwischen einem Endanwender und einem Informatiker, dass der Informatiker mit der Kommandozeile seines Betriebssystems umgehen kann. Seit Anfang der 80er Jahre gibt es graphische Benutzeroberflächen, die den Umgang mit dem Computer für Endanwender vereinfacht hat. Vorreiter war hier sicher zum einem die Firma Apple mit ihren Macintosh Rechnern aber auch andere Systeme stellten in den 80er Jahren intuitive graphische Benutzeroberflächen zur Verfügung, wie der Atari ST oder Commodore Amiga. Ebenso bekamen in dieser Zeit Unix Betriebssysteme komfortable graphische Benutzeroberflächen wie Suns Solaris und HP-UX.

Vor dieser Zeit war eine Benutzerinteraktion immer rein über die Tastatur und fast immer über eine Kommandozeile. Bis heute stellen alle Betriebssysteme einen direkten textuellen Zugang über eine Kommandozeile zur Verfügung. Eine Kommandozeile ist ein Programm, das Texteingaben erwartet und diese für bestimmte Aktionen im Betriebssystem ausführt. Hierzu gehört zum Beispiel, dass man über die Kommandozeile beliebige Programme starten kann. Auch Programme, die selbst wieder eine graphische Benutzeroberfläche haben. Alle Programme können aber auch selbst textuelle Ausgaben auf die Kommandozeile schreiben und Texteingaben von der Kommandozeile lesen.

Die Kommandozeile ist also in der Regel nicht die direkte Interaktion mit dem Betriebssystem, sondern selbst ein Programm, das die Benutzereingaben interpretiert und in Betriebssystembefehle umsetzt. Ein solches Terminalprogramm wird als *shell* bezeichnet. Wir werden im Folgenden einen rudimentären Umgang mit dem Programm *bash* zeigen. *bash* steht für *GNU Bourne-Again SHell*. Für Linux steht die *bash* standardmäßig als Kommandozeilenprogramm zur Verfügung. Auf Windows-Systemen gibt es Software-Pakete, die installiert werden können, so dass genauso auf Windows mit *bash* gearbeitet werden kann, wie auf Linux, zum Beispiel mit http://www.cygwin.com/www.cygwin.com.

Der Vorteil des Umgangs mit der Kommandozeile ist vielfältig. Es lassen sich zum einen hier viele Aufgaben unabhängig von anderen graphischen Programmen durch Eingabe weniger Befehle bewerkstelligen. Der Zugang zum Betriebssystem ist direkter. Dinge die einem graphische Programme nicht zeigen sondern verstecken, können eingesehen werden. Es lassen sich leicht Arbeitsschritte durch Skripte automatisieren, was bei graphischen Programmen so nicht möglich ist. Und vor allen Dingen, selbst wenn keine graphische Benutzeroberfläche mehr vorliegt, lassen sich mit der Kommandozeile alle Funktionen des Systems kontrollieren. Besonders schnell erkennt man den Nutzen der Kommandozeilen, wenn man einen Server über das Internet administrieren muss. Ein Zugang zu

[2] Und wer es wissen will: für eigene kleine Projekte benutze ich persönlich Scala oder Haskell, für Webanwendungen GWT oder das Scala Lift Framework,und für mein Smartphone Java oder Kotlin für Android.

einem Server steht über eine Kommandozeile immer zur Verfügung, eine graphische Benutzeroberfläche jedoch so gut wie nie.

Also gibt es Gründe genug, den Umgang mit Kommandozeilenbefehlen einzuüben und diesen Umgang im Laufe des Studiums immer weiter zu verbessern. Zum Glück ist es gar nicht so schwer und die wichtigsten Befehle sind schnell erlernt.

Das Modul »Einführung in die Informatik« bietet ausführlich Gelegenheit, den Umgang mit der Kommandozeile einzuüben. In Kontext unseres Moduls geben wir jetzt eine Schnelleinführung in die wichtigsten ersten Schritte, die zum Schreiben, Kompilieren und Ausführen von Java-Programmen notwendig sind.

1.3.1 Basisbefehle

Wenn man die Kommandozeile geöffnet hat, befindet diese sich immer in einem Ordner des Dateisystems. Alle Befehle, die eingegeben werden beziehen sich auf diesem Ordner. Er wird auch als Arbeitsverzeichnis bezeichnet.

`ls` Dateien Auflisten

Der erste Befehl, den wir vorstellen, dient dazu die Dateien des Arbeitsverzeichnisses aufzulisten. Er besteht nur aus zwei Buchstaben: `ls`. `ls` wird als *list* gesprochen. Geben wir den Befehl in der Kommandozeile aus, so bekommen wir eine Auflistung aller Dateien des Arbeitsverzeichnisses.

```
panitz@ThinkPad-T430:~/oose$ ls
panitz@ThinkPad-T430:~/oose$
```

In diesem Fall war die Auflistung etwas enttäuschend, denn es befindet sich keine Datei im Arbeitsverzeichnis. Wird der Befehl in einem anderen Arbeitsverzeichnis aufgerufen, in dem sich Dateien befinden, so werden diese tatsächlich aufgelistet:

```
panitz@ThinkPad-T430:~/fh/oose/v250107$ ls
build      build.properties~  build.xml~  docs  web
build.properties  build.xml       dist   src
panitz@ThinkPad-T430:~/fh/oose/v250107$
```

Es werden in diesem Fall 9 Dateien aufgelistet. Wenn wir mehr Informationen über diese Dateien erhalten wollen, so können wir dem Befehl `ls` einen zusätzlichen Befehlsparameter mitgeben. Solche beginnen in der Regel mit einem Minuszeichen. `ls` kennt den Parameter `-l`, der für *long* steht und eine ausführliche Auflistung der Dateien bewirkt.

```
panitz@ThinkPad-T430:~/fh/oose/v250107$ ls -l
insgesamt 60
drwxr-xr-x 3 panitz panitz  4096 Okt  6  2010 build
-rw-r--r-- 1 panitz panitz   253 Okt  6  2010 build.properties
-rw-r--r-- 1 panitz panitz   246 Okt  6  2010 build.properties~
-rw-r--r-- 1 panitz panitz 16334 Okt  6  2010 build.xml
-rw-r--r-- 1 panitz panitz 16336 Okt  6  2010 build.xml~
drwxr-xr-x 3 panitz panitz  4096 Okt  6  2010 dist
drwxr-xr-x 2 panitz panitz  4096 Okt  6  2010 docs
drwxr-xr-x 2 panitz panitz  4096 Okt  6  2010 src
drwxr-xr-x 3 panitz panitz  4096 Okt  6  2010 web
panitz@ThinkPad-T430:~/fh/oose/v250107$
```

Zusätzlich werden jetzt zu jeder Datei eine ganze Reihe weiterer Informationen angezeigt. Für jede Datei wird dazu eine Zeile ausgegeben. Zusätzliche Informationen sind die Benutzerrechte der Datei, die Dateigröße, das Datum der letzten Änderung und die Angabe, ob es sich bei der Datei um einen Ordner handelt.

Es gibt Dateien, die der Befehl `ls` normaler Weise nicht anzeigt, die sogenannten versteckten Dateien. Eine Datei gilt als versteckt, wenn ihr Name mit einem Punkt beginnt. Fügt man dem Befehl `ls` den Paramter `-a` hinzu, werden alle, auch die versteckten Dateien, aufgelistet.

```
panitz@ThinkPad-T430:~/oose$ ls
panitz@ThinkPad-T430:~/oose$ ls -a
.  ..  .versteckt
panitz@ThinkPad-T430:~/oose$ ls -a -l
insgesamt 20
drwxrwxr-x   2 panitz panitz  4096 Sep 18 11:27 .
drwxr-xr-x 286 panitz panitz 16384 Sep 16 14:16 ..
-rw-rw-r--   1 panitz panitz     0 Sep 18 11:27 .versteckt
panitz@ThinkPad-T430:~/oose$
```

Wie man hier sieht, wurde durch `ls` keine Datei aufgelistet. Der zusätzliche Parameter `-a` bewirkt, dass drei zusätzliche Dateien aufgelistet werden, die jeweils mit einem Punkt im Namen beginnen. `-a -l` zeigt, dass zwei dieser versteckten Dateien Ordner sind.

Auf der ursprünglich aus DOS stammenden Kommandozeile der Microsoft Betriebssysteme heißt der analoge Befehl zum Auflisten der Dateien `dir`.

`cd` Wechseln des Arbeitsverzeichnisses

Wir haben festgestellt, dass sich alle Befehle der Kommandozeile auf das Arbeitsverzeichnis beziehen. Da man natürlich nicht immer mit den Dateien eines festen Arbeitsverzeichnisses arbeiten möchte, gibt es einen Befehl zum Wechseln des Arbeitsverzeichnisses. Dieses ist der Befehl `cd`, der für *change directory* steht. Er bekommt als Argument den Ordner angegeben, in den man wechseln möchte.

```
panitz@ThinkPad-T430:~$
panitz@ThinkPad-T430:~$ cd oose
panitz@ThinkPad-T430:~/oose$
```

In diesen Beispiel wird in den Unterordner `oose` des aktuellen Arbeitsverzeichnisses gewechselt. Anschließend ist dieser das neue Arbeitsverzeichnis. Im Prompt der Kommandozeile, das ist die Meldung, mit der eine Eingabe erwartet wird, wird angezeigt, welches das aktuelle Arbeitsverzeichnis ist.

Es ist natürlich nicht nur möglich in einen bestimmten Unterordner zu wechseln, sondern man kann auch weiter in den übergeordneten Ordner wechseln. Dieser Ordner hat einen reservierten Namen, aus zwei Punkten.

```
panitz@ThinkPad-T430:~/oose$ cd ..
panitz@ThinkPad-T430:~$
```

Es gibt einen weiteren besonderen Ordner. Dieses ist das Heimatverzeichnis des Benutzers. In diesem Heimatverzeichnis startet auch standardmäßig die Kommandozeile. Dieses ist das Startverzeichnis, in dem der Benutzer seine eigenen Dateien speichert. Es wird mit dem Tildesymbol `~` bezeichnet.

```
panitz@ThinkPad-T430:~/oose$ cd ~
panitz@ThinkPad-T430:~$
```

In dieses Verzeichnis wechselt der Befehl `cd` auch, wenn kein Ordner angegeben wird, in dem gewechselt werden soll.

```
panitz@ThinkPad-T430:~/oose$ cd
panitz@ThinkPad-T430:~$
```

Es ist nicht nur möglich, relativ vom aktuellen Arbeitsverzeichnis in ein Unterverzeichnis oder das Elternverzeichnis zu wechseln, sondern man kann auch absolut von der Wurzel des Dateisystems aus einen Pfad in ein bestimmtes Verzeichnis angeben. Dann ist vollkommen egal, in welchen Arbeitsverzeichnis sich die Kommandozeile bei Ausführung des Befehls befindet. Ein absoluter Pfad beginnt mit einem Schrägstrich `/`, der die Wurzel des gesamten Dateisystems bezeichnet. Ausgehend von dieser kann nun eine Folge von Ordner angegeben werden, die durch einen Schrägstrich getrennt werden.

```
panitz@ThinkPad-T430:~$ cd /home/panitz/fh/oose/v250107/
panitz@ThinkPad-T430:~/fh/oose/v250107$
```

Ein Pfad muss aber wiederum nicht absolut von der Wurzel des Dateisystems beginnen, sondern kann auch relativ vom Arbeitsverzeichnis beginnen. Hierzu ist der erste Schrägstrich wegzulassen.

```
panitz@ThinkPad-T430:~/fh/oose/v250107$ cd ../../pmt/v270513/
panitz@ThinkPad-T430:~/fh/pmt/v270513$
```

So wie es mit den zwei Punkten die Möglichkeit gibt, das übergeordnete Verzeichnis anzugeben, kann man auch das aktuelle Verzeichnis angeben. Hierzu dient der einfache Punkt.

`ls` mit Pfadangaben

Nachdem wir nun die Pfadangaben für die Kommandozeile kennen gelernt haben, können wir diese auch für den Befehl `ls` anwenden. Bisher haben wir uns von `ls` immer nur eine Auflistung der Dateien des Arbeitsverzeichnisses geben lassen. Wir können aber durch Angabe eines Pfades, der einen bestimmten Ordner bezeichnet, `ls` dazu bringen, die Dateien in diesem Ordner anzuzeigen.

```
panitz@ThinkPad-T430:~/fh/pmt/v270513$ ls /home/panitz/fh/oose/vor130605/
Counter.class    GUI$3.class     Inner.java~
Counter.java    GUI.class     KnopfAktion.class
Counter.java~    GUI.java     KnopfAktion.java
EverySecond.class  GUI.java~     KnopfAktion.java~
EverySecond.java  Inner$1.class     MyGraphic.class
EverySecond.java~  Inner$1InnerInner.class  MyGraphic.java
GUI$1.class    Inner.class     MyGraphic.java~
GUI$1KnopfAktion.class  Inner$InnerInner.class
GUI$2.class    Inner.java
panitz@ThinkPad-T430:~/fh/pmt/v270513$
```

Sollte man nur an Informationen einer bestimmten Datei interessiert sein, so lässt sich dieses durch einen Pfad auf diese Datei,bestimmen.

```
panitz@ThinkPad-T430:~$ ls -l /home/panitz/fh/oose/GUI.java
-rw-r--r-- 1 panitz panitz 1111 Okt  6  2010 /home/panitz/fh/oose/GUI.java
panitz@ThinkPad-T430:~$
```

Innerhalb von Dateinamen hat der Stern `*` noch eine besondere Bedeutung. Er steht für eine beliebige Folge von beliebigen Buchstaben. So bedeutet z.B. `*.jpg` alle Dateinamen mit der Endung `.jpg`. So lassen sich Mengen von Dateien ansprechen. Im folgenden Beispiel werden alle Dateien, deren Name mit `GUI` beginnt und die die Endung `.class` haben.

```
panitz@ThinkPad-T430:~$ ls -l /home/panitz/GUI*.class
-rw-r--r-- 1 panitz panitz  875 Okt  6  2010 /home/panitz/GUI$1.class
-rw-r--r-- 1 panitz panitz  875 Okt  6  2010 /home/panitz/GUI$2.class
-rw-r--r-- 1 panitz panitz  585 Okt  6  2010 /home/panitz/GUI$3.class
-rw-r--r-- 1 panitz panitz 1655 Okt  6  2010 /home/panitz/GUI.class
panitz@ThinkPad-T430:~$
```

`mkdir` Neue Verzeichnisse Anlegen

Es lassen sich mit dem Befehl **mkdir** neue Verzeichnisse anlegen. Hierzu schriebt man hinter den Befehl als Pfadangabe den Namen des anzulegenden Verzeichnisses. Dieses kann wieder relativ zu dem Arbeitsverzeichnis sein oder auch eine absolute Pfadangabe.

```
panitz@ThinkPad-T430:~$ mkdir neuesTestVerzeichnis
panitz@ThinkPad-T430:~$ cd neuesTestVerzeichnis/
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$
```

Mit dem Befehl **rmdir** lassen sich Verzeichnisse wieder löschen. Hierzu muss das Verzeichnis allerdings leer sein. Es darf keine Unterverzeichnisse oder Dateien mehr enthalten.

```
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ cd ..
panitz@ThinkPad-T430:~$ rmdir neuesTestVerzeichnis/
panitz@ThinkPad-T430:~$
```

`touch` Neue Dateien Anlegen oder Dateien Aktualisieren

Es lassen sich auch neue leere Dateien von der Kommandozeile anlegen. Hierzu kann der Befehl `touch` benutzt werden. Ihm gibt man an, welche Datei neu angelegt werden soll.

```
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ ls -l
insgesamt 0
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ touch neueTestdatei
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ ls -l
insgesamt 0
-rw-rw-r-- 1 panitz panitz 0 Sep 19 10:33 neueTestdatei
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$
```

Sollte der Befehl für eine Datei aufgerufen werden, die bereits existiert, so wird der Dateiinhalt nicht verändert, aber das Datum der letzten Änderung aktualisiert.

```
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ ls -l
insgesamt 0
-rw-rw-r-- 1 panitz panitz 0 Sep 19 10:33 neueTestdatei
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ touch neueTestdatei
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ ls -l
insgesamt 0
-rw-rw-r-- 1 panitz panitz 0 Sep 19 10:34 neueTestdatei
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$
```

`cat`, `more` und `less` Dateiinhalte Anzeigen

Auch der Inhalt einer Textdatei lässt sich auf der Kommandozeile anzeigen. Hierzu kann der Befehl `cat` benützt werden. `cat` steht dabei für *concatenate*, also zum Konkatenieren, dem Aneinanderhängen von Dateiinhalten. Der Name kommt daher, dass dem Befehl `cat` nicht nur eine sondern eine Liste von Dateien angegeben werden kann. Es werden dann die Inhalte dieser Dateien nacheinander auf der Kommandozeile ausgegeben.

```
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ cat neueTestdatei
Hier steht nur ein kleiner Beispieltext.
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$
```

Ist eine Textdatei sehr groß, so flutscht die Ausgabe von `cat` sehr schnell auf der Kommandozeile vor unserem Auge vorbei. Man will einen Dateiinhalt in der Regel durchlesen. Hierzu gibt es die funktionsgleichen Programme `more` und `less`. Auch mit diesen Befehlen wird der Dateiinhalt auf der Kommandozeile ausgegeben. Allerdings immer nur ein Seite, die genau auf das Fenster der Kommandozeile passt. Nun kann man mit Drücken der Leertaste jeweils zur nächsten Seite gesprungen werden. Mit Drücken der Taste Q wird die Anzeige der Datei beendet.

`pwd` Pfadangabe

Man kann durch den Befehl `pwd` sich den absoluten Pfad zum Arbeitsverzeichnis anzeigen lassen.

```
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ pwd
/home/panitz/neuesTestVerzeichnis
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$
```

`rm` Dateien Löschen

Dateien lassen sich auch löschen. Hierzu dient der Befehl `rm`, der memotechnisch für *remove* steht. (Auf der Kommandozeile für Microsoft Betriebssysteme heißt der entsprechende Befehl `del`.) Dieser Befehl ist nichts für Feiglinge, denn die mit ihm gelöschten

Dateien sind unwiederbringlich weg. Sie sind nicht in einem Papierkorb zwischengelagert, aus dem sie wieder reaktiviert werden können. Besonders gefährlich kann es sein, wenn dieser Befehl mit einer Dateiangabe aufgerufen wird, die über das Sternsymbol mehrere Dateien bezeichnet. `rm *.jpg *.gif *.png` löscht zum Beispiel alle Bilddateien im Arbeitsverzeichnis. `rm *` löscht alle Dateien im Arbeitsverzeichnis. Der Befehl kennt dann noch den Parameter `-r`. Er steht dafür, dass im Falle eines Ordners zunächst alle Dateien und Unterordner dieses Ordners zu löschen sind und dann der Ordner selbst. So löscht also `rm -r *` alle Dateien und alle Ordner komplett aus dem Arbeitsverzeichnis. In Ihrem Heimatverzeichnis aufgerufen löschen Sie damit also alle Ihre Dateien. Also Vorsicht!

`mv` Dateien Umbenennen und Verschieben

Der Befehl `mv` steht memotechnisch für *move*. Er dient dazu, eine Datei an einen anderen Ort, also in einen anderen Ordner, zu verschieben. Er hat aber auch eine zweite verwandte Funktionalität. Er dient auch dazu eine Datei umzubenennen. Mindestens zwei Parameter sind diesem Befehl anzugeben. Die Datei, die verschoben oder umbenannt werden soll, und das Ziel wohin diese verschoben werden soll. Ist das Ziel ein existierender Ordner im Dateisystem, so wird die Datei dahin verschoben. Ist es ein Dateiname eventuell einer noch nicht existierenden Datei, dann wird die Datei umbenannt.

```
panitz@ThinkPad-T430:~/oose$ ls
testdatei.txt
panitz@ThinkPad-T430:~/oose$ mv testdatei.txt neuerName.txt
panitz@ThinkPad-T430:~/oose$ ls
neuerName.txt
panitz@ThinkPad-T430:~/oose$ mkdir einNeuerOrdner
panitz@ThinkPad-T430:~/oose$ mv neuerName.txt einNeuerOrdner/
panitz@ThinkPad-T430:~/oose$ ls
einNeuerOrdner
panitz@ThinkPad-T430:~/oose$ cd einNeuerOrdner/
panitz@ThinkPad-T430:~/oose/einNeuerOrdner$ ls
neuerName.txt
panitz@ThinkPad-T430:~/oose/einNeuerOrdner$
```

Auch hier muss man ein wenig Vorsicht walten lassen. Existiert die Zieldatei bereits, so wird sie mit der Quelldatei überschrieben und damit quasi gelöscht.

Man kann den Befehl `mv` auch nutzen, um mehrere Dateien in einen Ordner zu verschieben. Dann erhält der Befehl mehr als zwei Argumente. Das letzte Argument bezeichnet dabei einen Ordner, in den die vorherigen Argumente zu verschieben sind. Auch hier kann man sich wieder des Sterns bedienen. So bewirkt `mv *.jpg meineBilder`, dass alle Dateien des Arbeitsverzeichnisses mit der Endeung `jpg` in den Unterordner `meineBilder` verschoben werden.

`cp` Dateien Kopieren

Sehr ähnlich, wie der Befehl `mv` funktioniert der Befehl `cp`, der dazu dient eine oder mehrere Dateien zu kopieren. Es handelt sich dazu im Prinzip um den gleichen Befehl wie `mv`, nur wird die Quelldatei in diesem Fall nicht gelöscht. Damit gibt es anschließend also zwei unabhängige Versionen der Quelldatei.

`man` Handbuchseiten

Wie kann man sich alle die Befehle und ihre Benutzung merken. Alle die kleinen Parameter. Die Antwort ist: RTFM. Dieses steht für den Satz *read the fucking manual* also eine etwas saloppe Aufforderung, das Handbuch zu lesen. Mit dem Befehl `man` können die Handbucheinträge aller Befehle aufgerufen werden. Wer z.B. nicht mehr weiß, wie der Befehl `ls` genau funktioniert, kann das mit `man ls` erfragen und bekommt folgende Ausgabe:

```
panitz@ThinkPad-T430:~/neuesTestVerzeichnis$ man ls

LS(1)                          User Commands                          LS(1)

NAME
    ls - list directory contents

SYNOPSIS
    ls [OPTION]... [FILE]...

DESCRIPTION
    List  information  about  the FILEs (the current directory by default).
    Sort entries alphabetically if none of -cftuvSUX nor --sort  is  speci-
    fied.

    Mandatory  arguments  to  long  options are mandatory for short options
    too.

    -a, --all
           do not ignore entries starting with .

    -A, --almost-all
           do not list implied . and ..

    --author
 Manual page ls(1) line 1 (press h for help or q to quit)
```

Wie sie sehen, gibt es mehr Optionen für diesen Befehl, als in diesem Skript angegeben. Das gilt für alle der hier vorgestellten Befehle.

Es lässt sich auch die Hilfe für die gesamte Kommandozeilenbenutzung anzeigen durch `man bash`.

1.3.2 Nützliche Standardprogramme

Im letzten Abschnitt haben wir die wichtigsten Befehle der Kommandozeile zum Umgang mit dem Dateisystem kennen gelernt. Prinzipiell kann jedes Programm über die Kommandozeile gestartet werden. Viele Programme haben selbst gar keine graphische Benutzerschnittstelle sondern sind zur Benutzung über die Kommandozeile vorgesehen, oder werden von anderen Programmen aus gestartet. In diesem Abschnitt sollen ein paar elementare Programme, die sehr nützlich sind, kurz vorgestellt werden.

Sicheres Einloggen auf anderen Rechner mit `ssh`

Das Programm `ssh` dient dazu, um sich auf einem Rechner über ein Netzwerk einzuloggen und quasi die dortige Kommandozeile zu benützen. Dieses ist natürlich besonders nützlich, wenn man einen Webserver warten möchte. Hierzu ist beim Aufruf von `ssh` der Rechnername des Rechners, auf den man sich einloggen möchte, als Argument anzugeben. Wir haben bei uns im Studienbereich den Rechner mit der Adresse `login1.cs.hs-rm.de`, auf dem Sie sich mit Ihrem Passwort einloggen können. Um anzugeben, als welcher Benutzer Sie sich auf dem Rechner einloggen wollen, können Sie Ihren Benutzernamen mit dem Klammeraffensymbol getrennt dem Rechnernamen voranstellen.

```
panitz@ThinkPad-T430:~/oose$ ssh panitz@login1.cs.hs-rm.de
panitz@login1.cs.hs-rm.de's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Oct  1 12:10:22 2013 from vpn501.hotsplots.net
panitz@fozzie(~)$
```

Jetzt arbeiten Sie auf dem Server. Dieses erkennen Sie zum Beispiel, wenn Sie jetzt den Befehl `ls` eintippen, denn dann stellen Sie fest, dass Sie die Dateien an der Hochschule und nicht auf Ihrem Rechner daheim sehen.

Wollen Sie die Verbindung mit dem fremden Rechner beenden, so können Sie das durch den Befehl `exit`:

```
panitz@fozzie(~)$ exit
Abgemeldet
Connection to login1.cs.hs-rm.de closed.
panitz@ThinkPad-T430:~/oose$
```

Transfer von Dateien mit `sftp`

Mit `ssh` können Sie auf die Kommandozeile eines Rechners über ein Netzwerk zugreifen. Manchmal will man aber eine oder mehrere Dateien von einem anderen Rechner holen oder auf einen anderen Rechner hoch laden. Hierzu dient das Programm `sftp` *(secure file transfer program)*. Auch dieses wird mit Ihrem Benutzernamen gefolgt von der Adresse des Servers gestartet. Wenn Sie in den Server eingeloggt sind, sind sie in einer Umgebung, in der Sie Befehle zum Dateitransfer abschicken können. Die Befehle heißen `put`, um Dateien auf den Server hochzuladen und `get` um Dateien von dem Server herunterzuladen. Mit dem aus der Kommandozeile bekannten Befehl `cd` können Sie durch das Dateisystem des Servers navigieren.

```
panitz@ThinkPad-T430:~/oose/einNeuerOrdner$ sftp panitz@login1.cs.hs-rm.de
panitz@login1.cs.hs-rm.de's password:
Connected to login1.cs.hs-rm.de.
sftp> put existierendeDatei
Uploading existierendeDatei to /home/staff/panitz/existierendeDatei
existierendeDatei                             100%    0     0.0KB/s   00:00
sftp> get Main.hs
Fetching /home/staff/panitz/Main.hs to Main.hs
sftp> exit
panitz@ThinkPad-T430:~/oose/einNeuerOrdner$
```

Auch dieses Programm wird durch die Eingabe des Befehls `exit` wieder beendet.

Kopieren über Rechnergrenzen mit `scp`

Statt mit dem Programm `sftp` können Sie auch mit der Anweisung von einem auf einen anderen Rechner kopieren. Ziel oder Quelle dieses Kopierbefehls können auf einem entfernten Rechner lliegen. Hierzu wird dem Rechnernamen des entfernten Rechners nach einem Doppelpunkt der Pfad im dortigen Dateisystem angegeben. Auch hier wird vor dem Rechnernamen mit dem Klammeraffen getrennt der Anwendername auf diesem Rechner angegeben.

```
$ scp -P 2200 target/LectureNotes-1.0-SNAPSHOT.war panitz@procomp.cs.hs-rm.de:~
panitz@procomp.cs.hs-rm.de's password:
LectureNotes-1.0-SNAPSHOT.war              49% 8336KB  62.7KB/s   02:17 ETA
```

Dateityp erfragen mit `file`

Eigentlich sollte der Typ einer Datei durch die Dateiendung korrekt angegeben sein. Manchmal fehlt aus unerfindlichen Gründen diese Endung, oder aber, Sie wurde falsch angegeben (Das passiert immer wieder bei Abgaben von studentischen Lösungen). Oder aber man kennt die Endung nicht. Dann ist das Programm `file` recht nützlich, denn es versucht herauszubekommen, um was für eine Datei es sich handelt.

```
panitz@ThinkPad-T430:~$ file FKT_Programm_11_13_V1_Montageflaechen-1
FKT_Programm_11_13_V1_Montageflaechen-1: PDF document, version 1.4
panitz@ThinkPad-T430:~$
```

Dateien in ein Archiv verpacken mit `tar`

Zur Archivierung oder dem Versand von einer großen Menge von Dateien, zum Beispiel aller Ihrer Urlaubsbilder, empfiehlt es sich, diese in einer einzigen Archivdatei zu bündeln und am besten noch zu komprimieren. Hierzu gibt es mehrere Programme und daraus resultierende Dateiformate wie zum Beispiel das Programm `zip`. Ein Standardprogramm auf der Unix-Welt ist hierzu das Programm `tar`. Es steht für *tape archive* und deutet so im Namen auf die Zeiten hin, als Daten auf Tonbändern archiviert wurden. Dem Programm `tar` wird durch eine Folge vom Buchstaben beim Aufruf angegeben, was es machen soll, z.B. ob es ein neues Archiv erzeugen oder ob es ein bestehendes entpacken soll. Dann folgen die Dateien, mit denen gearbeitet werden soll. Das Erzeugen eines neuen Archivs wird durch den Buchstaben `c` für *create* angegeben. Mit dem Buchstaben `f` wird angegeben, dass jetzt der Name der Archivdatei folgt. Will man also alle Bilddateien im Arbeitsverzeichnis in eine Archivdatei `bilder.tar` verpacken, so geht das mit dem Befehl
`tar cf bilder.tar *.jpg`.

Zum Entpacken einer Archivdatei benützt man den Buchstaben `x` für *extract* statt des Buchstabens `c`. Die mit obigen Befehle erzeugte Archivdatei lässt sich also mit
`tar xf bilder.tar` wieder entpacken.

Will man sich nur eine Auflistung der Dateien, die in einer Archivdatei verpackt wurden, geben lassen, so nimmt man den Buchstaben `t` für *table*. Fügt man noch den Buchstaben `v` für *verbode* hinzu, so gibt das Programm etwas mehr Ausgaben. Soll die Archivdatei komprimiert werden, so füge man noch den Buchstaben `z` hinzu.

Hier ein kleines Beispiel zum Verpacken, Auflisten und Entpacken einer kleinen Archivdatei:

```
panitz@ThinkPad-T430:~/Bilder$ tar cvzf bilder.tgz *.png
Bildschirmfoto1.png
Bildschirmfoto2.png
Bildschirmfoto3.png
Logo-gross.png
tree.png
panitz@ThinkPad-T430:~/Bilder$ tar tvzf bilder.tgz
-rw-rw-r-- panitz/panitz 368235 2013-07-17 19:05 Bildschirmfoto1.png
-rw-rw-r-- panitz/panitz 157235 2013-08-01 19:49 Bildschirmfoto2.png
-rw-rw-r-- panitz/panitz 156051 2013-08-01 21:59 Bildschirmfoto3.png
-rw-r--r-- panitz/panitz  14209 2009-01-03 14:47 Logo-gross.png
-rw-rw-r-- panitz/panitz   3733 2013-05-14 15:25 tree.png
panitz@ThinkPad-T430:~/Bilder$ tar xvzf bilder.tgz
Bildschirmfoto vom 2013-07-17 19:05:30.png
Bildschirmfoto vom 2013-08-01 19:49:16.png
Bildschirmfoto vom 2013-08-01 21:59:44.png
Logo-gross.png
tree.png
panitz@ThinkPad-T430:~/Bilder$
```

Weitere Tipps und Tricks

Den ersten und rudimentären Umgang mit der Kommandozeile haben wir eingeübt.

Arbeiten mit Vervollständigung und Befehlshistorie Die Kommandozeile merkt sich die bereits ausgeführten Befehle. Oft will man einen ähnlichen oder den gleichen Befehl noch einmal ausführen lassen. Die Pfeiltasten liefern heute eine einfache Möglichkeit durch die Befehlshistorie zu navigieren und zuvor bereits ausgeführte Befehle wieder anzeigen zu lassen.

Auch die Tabulaturtaste vereinfacht die Arbeit mit der Kommandozeile und hilft dabei, weniger eintippen zu müssen. Wenn von der Kommandozeile der Name einer Datei erwartet wird, kann man die ersten Buchstaben der Datei tippen und dann mit Hilfe der Tabulatortaste sich den kompletten Dateinamen vervollständigen zu lassen. So müssen lange Dateinamen nicht komplett eingetippt werden.

Umleiten der Ausgabe in eine Datei Wenn man einen Befehl startet, der eine ausführliche Ausgabe auf der Kommandozeile macht, dann kann man diese auf einfache Weise von der Kommandozeile in eine Datei schreiben lassen. Hierzu bedient man sich des Größer-Zeichens. Nach dem Befehl wird das Größer-Zeichen und der Dateiname geschrieben. Dieses bewirkt, dass die Ausgabe des Befehls nicht mehr auf der Kommandozeile angezeigt wird, sondern in die angegebene Datei geschrieben wird.

Zwei Programme verbinden Ein besonders eleganter Trick ist es, die Ausgabe eines Befehls direkt wieder als Eingabe für einen weiteren Befehl zu benützen. Hierzu dient die sogenannte *pipe*, die durch einen vertikalen Strich bezeichnet wird.

Ist das Arbeitsverzeichnis zum Beispiel ein Ordner, in dem sehr viele Dateien liegen, dann führt der Befehl `ls -l` zu einer umfangreichen Ausgabe auf der Kommandozeile, die schnell auf dem Bildschirm vorbei rauscht. Jetzt kann man die Ausgabe von `ls -l` direkt an das Programm `less` weiterleiten, indem man die Ausgabe von `ls` dem Befehl `less` über eine *pipe* direkt als Eingabe gibt. Der gesamte Aufruf lautet dann `ls -l | less`.

1.3.3 Erste Java Programme auf der Kommandozeile

In diesem Abschnitt soll gezeigt werden, wie prinzipiell Programme auf der Kommandozeile entwickelt werden. Wir erinnern uns, dass ein Programm zunächst einmal nichts weiter als ein Text ist. Wir müssen also einen Text schreiben. Dieser Text ist dann von einem besonderen Programm in eine Maschinensprache zu übersetzen. Ein solche Programm heißt Kompilator *(eng. compiler)*. Der Kompilator untersucht den Quelltext und übersetzt diesen, sofern das möglich ist, in eine oder mehrere binäre Dateien, die Befehle für eine Maschine codieren. Schließlich müssen diese binären Dateien zur Ausführung auf einer Maschine gebracht werden.

Quelltext editieren

Wir benötigen ein Programm, das uns erlaubt, eine Quelltextdatei zu erstellen. Ein solches Programm nennt man einen Texteditor. Dieser ist nicht zu verwechseln mit einem Textverarbeitungsprogramm, in dem nicht nur Text, sondern auch ein Layout mit Schriftarten, Schriftgrößen etc. erstellt wird. Es gibt tatsächlich Programme, die dieses alleine auf der Kommandozeile ohne eine graphische Benutzerpberfläche erlauben. Der Standardeditor für die Kommandozeile heißt `vi`. Es gibt sogar auch heute noch Leute, die nur mit Hilfe des `vi` ihre Programme schreiben. Aber gerüchteweise sollen das nur noch fusselige, bärtige, nuschelnde, selbstdrehende Systemadministratoren sein.

Trotzdem sollt jeder Informatiker den rudimentären Umgang mit dem Programm `vi` kennen, denn irgendwann in seiner Laufbahn wird der Moment kommen, an dem er schnell eine Konfigurationsdatei ändern muss, aber nur einen Kommandozeilenzugang zu dem Rechner hat.

Das Programm `vi` kann gestartet werden mit dem Namen der Datei, die man editieren möchte. Existiert keine Datei mit diesen Namen, so wird die Datei dann neu angelegt. Wenn wir also eine Javaquelltextdatei mit dem Dateinamen `FirstProgram.java` editieren und erstellen möchten, so starten wir den `vi` mit dem Befehl:

```
vi FirstProgram.java
```

Das Programm `vi` hat zwei Modi. Den Befehlsmodus, in dem jede Eingabe als Befehl für den Editor interpretiert wird, und den Einfügemodus, in dem jede Eingabe als Text für den Inhalt der Datei interpretiert wird. Der Befehlsmodus ersetzt die Menus, die aus den Texteditoren mit graphischer Benutzerführung bekannt sind.

Vom Befehlsmodus kann man in den Einfügemodus durch Eingabe von `i` für *insert* oder `a` für *append* wechseln. Nach der Eingabe des Befehls `i` befindet man sich in Einfügemodus und alle folgenden Buchstaben werden vor der markierten Stelle des Dokuments eingefügt. Nach dem Befehl `a` werden beim Einfügemodus alle folgenden Buchstaben nach der ursprünglich markierten Stelle eingefügt.

Vom Einfügemodus in den Befehlsmodus wechselt man durch Drücken der `ESC`-Taste.

Die wichtigsten Befehlsfolgen im Befehlsmodus betreffen natürlich das Speichern der Datei und auch das Verlassen des Programms. Diese Befehlsfolgen werden durch den Doppelpunkt eingeleitet. Der Befehl `:wq` bewirkt im Befehlsmodus, dass die Datei gespeichert wird (`w` für *write*) und das Programm `vi` verlassen wird (`q` für *quit*).

Versuchen Sie also, Ihre erste Javaquelltextdatei mit dem `vi` zu schreiben. Jede Javaquelltextdatei hat die Endung `.java`. In einer Javaquelltextdaei wird jeweils genau eine sogenannte Klasse definiert. Folgendes ist die kleinste mögliche Klasse. Sie heißt `FirstProgram` und ist entsprechend in eine Datei `FirstProgramm.java` zu speichern.

FirstProgramm.java

```
class FirstProgramm{
}
```

Es gibt noch eine ganze Reihe weiterer Texteditoren, die auf der Kommandozeile benutzt werden können. Ich persönlich benutze wann immer möglich das Programm `emacs`, welche mit einer Menuführung daherkommt, aber auch einen Modus hat, der ohne graphische Benutzeroberfläche auskommt. Hierzu startet man den `emacs` mit dem Argument `-nw` (für *no window*).

Ein weiteres solcher Texteditor ist das Programm `joe`. Allerdings sind `emacs` oder `joe` nicht auf allen Systemen installiert, wohingegen man eigentlich immer davon ausgehen kann, dass `vi` existiert.

Quelltext kompilieren

Nun brauchen wir den Kompilator, der einen Quelltext in einen binären Maschinencode überführt. Für Java heißt das entsprechende Programm `javac`. Durch den Aufruf des Programms `javac` kann man sich davon überzeugen, dass die Java-Entwicklerumgebung auf dem Rechner installiert ist. Ruft man den Kompilator auf, so macht er folgende Ausgabe, die über seine Benutzung informiert:

```
panitz@ThinkPad-T430:~$ javac
Usage: javac <options> <source files>
where possible options include:
 -g                         Generate all debugging info
 -g:none                    Generate no debugging info
 -g:{lines,vars,source}     Generate only some debugging info
 -nowarn                    Generate no warnings
 -verbose                   Output messages about what the compiler is doing
 -deprecation               Output source locations where deprecated APIs ar
 -classpath <path>          Specify where to find user class files and annot
 -cp <path>                 Specify where to find user class files and annot
 -sourcepath <path>         Specify where to find input source files
 -bootclasspath <path>      Override location of bootstrap class files
 -extdirs <dirs>            Override location of installed extensions
 -endorseddirs <dirs>       Override location of endorsed standards path
 -proc:{none,only}          Control whether annotation processing and/or com
 -processor <class1>[,<class2>,<class3>...] Names of the annotation processo
 -processorpath <path>      Specify where to find annotation processors
 -d <directory>             Specify where to place generated class files
 -s <directory>             Specify where to place generated source files
 -implicit:{none,class}     Specify whether or not to generate class files f
 -encoding <encoding>       Specify character encoding used by source files
 -source <release>          Provide source compatibility with specified rele
 -target <release>          Generate class files for specific VM version
 -version                   Version information
 -help                      Print a synopsis of standard options
 -Akey[=value]              Options to pass to annotation processors
 -X                         Print a synopsis of nonstandard options
 -J<flag>                   Pass <flag> directly to the runtime system
 -Werror                    Terminate compilation if warnings occur
 @<filename>                Read options and filenames from file

panitz@ThinkPad-T430:~$
```

Wie man dieser Ausgabe entnehmen kann, gibt es das Argument `-version`, mit dem die Versionsnummer des Kompilators erfragt werden kann.

```
panitz@ThinkPad-T430:~$ javac -version
javac 11.0.4
panitz@ThinkPad-T430:~$
```

In diesem Fall handelt es sich also um den Java-Kompilator für Java der Version 11.0.4

Jetzt können wir für die Java-Quelltextdatei `FirstProgram.java` durch den Kompilator in eine binäre Datei übersetzen lassen. Hierzu wird der Kompilator mit dem Dateinamen der Quelltextdatei als Argument aufgerufen. Das Ergebnis ist im Erfolgsfall eine neu generierte Datei, die `class`-Datei.

```
panitz@ThinkPad-T430:~/oose$ ls
FirstProgram.java
panitz@ThinkPad-T430:~/oose$ javac FirstProgram.java
panitz@ThinkPad-T430:~/oose$ ls
FirstProgram.java  FirstProgram.class
panitz@ThinkPad-T430:~/oose$
```

Wie man sieht, wurde die Quelltextdatei erfolgreich übersetzt und eine Class-Datei erzeugt. Sehr oft ruft man den Kompilator auf, doch statt den Quelltext zu übersetzen, bricht er mit einer Fehlermeldung ab. Dieses ist der Fall, wenn in der Quelltextdatei kein gültiges Java-Programm geschrieben ist. In unseren Fall war aber der Übersetzungsvorgang erfolgreich.

Programme ausführen

Schließlich soll die binäre Datei mit den Maschinenbefehlen, ausgeführt werden. Im Falle von Java handelt es sich dabei nicht um eine real als Hardware existierende Maschine, sondern um eine gedachte, eine sogenannte virtuelle Maschine. Diese wird durch ein Programm realisiert. Daher braucht man zum Ausführen von Javaprogrammen das Programm, das die virtuelle Maschine realisiert. Dieses Programm heißt sinniger Weise `java`. Auch in diesem Fall kann man sich durch den Befehl `java` auf der Kommandozeile davon überzeugen, dass die virtuelle Maschine auf dem Rechner installiert ist. Das Programm `java` wird auch als Javainterpreter bezeichnet.

Der Javainterpreter wird aufgerufen mit dem Namen der Klasse, die ausgeführt werden soll. Hierbei wird kein Dateiname angegeben, auch keine Dateiendung, sondern nur der Name der Klasse, in unserem Fall `FirstProgram`. Somit ist der Befehl zum Ausführen der Klasse `java FirstProgram`.

Rufen wir dieses auf, so stellen wir fest, dass der Javainterpreter eine Fehlermeldung ausgibt:

```
panitz@ThinkPad-T430:~/oose$ java FirstProgram
Fehler: Hauptmethode in Klasse FirstProgram nicht gefunden.
            Definieren Sie die Hauptmethode als:
   public static void main(String[] args)
panitz@ThinkPad-T430:~/oose$
```

In diesem Fall ist die Meldung sogar auf Deutsch. Sie besagt, dass der Klasse etwas fehlt, nämlich eine sogenannte Hauptmethode, in der die Javabefehle stehen, die ausgeführt werden sollen. Die Fehlermeldung gibt sogar an, wie eine solche Hauptmethode auszusehen hat. Schreiben wir jetzt einmal eine zweite Klasse, die eine solche Hauptmethode beinhaltet:

SecondProgram.java

```
class SecondProgram{
  public static void main(String[] args){
  }
}
```

Übersetzen wir diese Klasse mit dem Kompilator und interpretieren sie mit dem Javainterpreter, so gibt es keine Fehlermeldung mehr. Allerdings passiert auch nichts, weil der innerhalb der Hauptmethode keine Befehle stehen, die ausgeführt werden sollen.

```
panitz@ThinkPad-T430:~/oose$ javac SecondProgram.java
panitz@ThinkPad-T430:~/oose$ java SecondProgram
panitz@ThinkPad-T430:~/oose$
```

Der erste und einfachste Befehl, mit dem man ein Programm dazu bringen kann, eine Rückmeldung an den Anwender zu geben, ist die Ausgabe eines Textes auf der Kommandozeile. In Java ist dieser Befehl: `System.out.println()`. In die runden Klammern ist der Text zu schreiben, der ausgegeben werden soll. Text wird dabei in Java in doppelten Anführungszeichen gesetzt.

Die folgende Javaklasse hat eine Hauptmethode innerhalb derer zweimal eine Zeile mit Text auf die Kommandozeile ausgegeben wird:

ThirdProgram.java

```
class ThirdProgram{
  public static void main(String[] args){
    System.out.println("Hallo Ilja!");
    System.out.println("Hallo Welt!");
  }
}
```

Kompilieren wir diese Klasse und interpretieren sie, so haben wir folgendes Verhalten:

```
panitz@ThinkPad-T430:~/oose$ javac ThirdProgram.java
panitz@ThinkPad-T430:~/oose$ java ThirdProgram
Hallo Ilja!
Hallo Welt!
panitz@ThinkPad-T430:~/oose$
```

1.4 Interaktives Java mit der jshell

Im letzten Abschnitt haben wir gesehen, wie man auf der Kommandozeile ein Javaprogramm schreibt, kompiliert und ausführt. War Ihnen das alles zu kompliziert für den

Einstieg. Dann gibt es eine schöne einfache Einstiegshilfe. Java hat auch einen interaktiven Mode, die `jshell`. Die `jshell` ist ein Interpreter für Java, Sie erwartet eine Eingabe, wertet diese zu einem Ergebnis aus und zeigt dieses Ergebnis als Ausgabe an. Man spricht dabei auch von einer *Read-Eval-Print-Loop* kurz REPL. Manche Programmiersprachen arbeiten allein mit einem solchen Interpreter, der einen direkten interaktiven Modus darstellt. Die Programmiersprache Python ist hierfür ein Beispiel. Manche Programmiersprachen bieten nur einen Compiler an, der den Quelltext in ein ausführbares Programm übersetzt. Die Programmiersprache C ist eine Beispiel hierfür. Und Manche Programmiersprachen bieten beides an: eine interaktive REPL und einen Compiler. Java zählt hierzu, aber fast alle modernen Programmiersprachen.

Die REPL von Java kann auf der Kommandozeile geöffnet werden mit dem Aufruf von `jshell`. Es öffnet sich die REPL, die eine Eingabe erwartet. Die einfachste Eingabe ist dabei wahrscheinlich eine Zahl.

```
panitz@panitz-ThinkPad-T430:~$ jshell
|  Welcome to JShell -- Version 11.0.4
|  For an introduction type: /help intro

jshell> 1
$1 ==> 1

jshell>
```

Das Ergebnis der Eingabe der Zahl 1 ist eben der Wert 1.

Als zweiten Schritt können wir anfangen Rechnungen durchführen zu lassen.

```
jshell> 1+1
$2 ==> 2

jshell> 17+4*2
$3 ==> 25

jshell>
```

Auf diese Art und Weise können wir spielerisch die ersten Schritten in Java machen und ein paar Java-Konstrukte kennenlernen. Bevor wir im Semester schrittweise systematisch die unterschiedlichen Konstrukte von Java kennenlernen, wollen wir zunächst einen intuitiven schnellen Einstieg mit der jshell wagen. Öffnen Sie also die jshell und spielen die folgenden Beispiele nach und machen weitere eigene Eingaben.

1.4.1 Ausdrücke auf Zahlen

Wir haben bereits mit den ersten einfachen Rechnungen auf Zahlen begonnen. Die einfachste Rechnung ist die, in der nur eine konstante Zahl berechnet wird.

```
~$ jshell
|  Welcome to JShell -- Version 11.0.4
|  For an introduction type: /help intro

jshell> 42
$1 ==> 42
```

Das Ergebnis Jeder Rechnung, die in der jshell durchgeführt wird, wird in einer Variablen gespeichert. Die Variablen werden durchnumeriert und beginnen mit dem Symbol $. Die jshell hat die Anweisung `/vars`, mit der sich alle Variablen und deren gespeicherte Werte anzeigen lassen:

```
jshell> /vars
|    int $1 = 42
```

Wie man sieht, gibt es jetzt die Variable `$1` mit dem Wert 42.

Geben wir eine Rechnung mit Addition und Multiplikation ein. So wird diese ausgewertet.

```
jshell> 17+4*2
$2 ==> 25
```

Und das Ergebnis in einer weiteren Variablen gespeichert.

```
jshell> /vars
|    int $1 = 42
|    int $2 = 25
```

Wir können in der jshell die Werte von gespeicherten Variablen wieder Abfragen. Hierzu wir nur der Variablenname eingegeben.

```
jshell> $1
$1 ==> 42
```

Ausdrücke, die ausgerechnet werden, können durch Klammern gruppiert sein. Damit lässt sich die Punkt- vor Strichrechnungsregel aus vorigen Beispiel umgehen.

```
jshell> (17+4)*2
$3 ==> 42
```

Wir haben bisher nur mit ganzen Zahlen gerechnet. Um eine Rechnung auf Kommazahlen zu machen, muss eine beteiligte Zahl eine Kommazahl sein.

```
jshell> (17+4)*2.0
$4 ==> 42.0
```

Wenn wir uns jetzt die gespeicherten Variablen anschauen erkennen wir, dass es zwei Arten von Zahlen zu geben scheint:

```
jshell> /vars
|    int $1 = 42
|    int $2 = 25
|    int $3 = 42
|    double $4 = 42.0
```

Die mit `int` markierten Variablen speichern ganze Zahlen, die mit `double` markierte Variable speichert eine Kommazahl. Dieses sind die erste beiden Datentypen, die Sie sehen. In Java hat jede Variable eine fest definierten Datentypen, der nicht verändert werden darf und von Anfang an fest steht. Man sagt: Java ist statisch getypt. Die Programmiersprache Python hingegen hat diese Eigenschaft nicht. Python ist eine dynamisch getypte Sprache.

Die Unterscheidung von ganzen Zahlen und Kommazahlen wird bei der Division relevant:

```
jshell> 17/4
$5 ==> 4
```

Wie man sieht erhält man eine ganze Zahl als Ergebnis. Erst wenn ein beteiligter Operand eine Kommazahl war, erhalten wir das Ergebnis mit der Nachkommazahl.

```
jshell> 17/4.0
$6 ==> 4.25
```

Wenn man an den Rest einer Division interessiert ist, dann kann man sich diesen mit dem Operanden % errechnen lassen:

```
jshell> 17%4
$7 ==> 1
```

Die 4 passt vier Mal in die 17 und es bleibt dann ein Rest von 1. Diese Rest-Operation der Division wird als Modulo-Operation bezeichnet.

Sie funktioniert auch auch Kommazahlen, auch wenn sie da weniger sinnvoll ist.

```
jshell> 17%4.0
$8 ==> 1.0
```

Mit diesen beiden Divisions-Operation lassen sich zum Beispiel die ersten und die letzten Stellen einer Zahl in Dezimalschreibweise leicht ermitteln.

Die Rechnung %100 ergibt die Zahl aus den letzten beiden Ziffern:

```
jshell> 1967%100
$9 ==> 67
```

Die Rechnung /100 ergibt die Zahl ohne den letzten beiden Ziffern:

```
jshell> 1967/100
$10 ==> 19

jshell>
```

1.4.2 Arbeiten mit Variablen

Das Ergebnis jeder Rechnung wird in der jshell in einer Variablen gespeichert. Diese sind durchnummeriert. Wenn Ihnen das nicht passt, können Sie aber auch explizit eigene Variablen für das Ergebnis anlegen:

```
jshell> var x = 42
x ==> 42
```

Variablen können in weiteren Rechnungen verwendet werden:

```
jshell> var x2 = 2*x+17
x2 ==> 101
```

Bestehende Variablen können auch einen neuen Wert bekommen:

```
jshell> x = -1
x ==> -1
```

Und wer will kann den Wert einer Variablen in einer Rechnung verwenden und anschließend das Ergebnis als neuen Wert für dieselbe Variable verwenden.

```
jshell> x = -x*17
x ==> 17
```

1.4.3 Vergleiche und Wahrheitswerte

Wir können Zahlenwerte über ihre Größe vergleichen. Die folgende Berechnung vergleicht, ob der in x gespeicherte größer oder gleich dem in x2 gespeicherten Wert ist.

```
jshell> x >= x2
$17 ==> false
```

Wenn wir uns in der jshell den Wert der Variable `$17` anzeigen lassen, lernen wir einen dritten Datentyp kennen.

```
jshell> /var $17
|    boolean $17 = false
```

Der Typ `boolean` steht für Wahrheitswerte. Es gibt die beiden Wahrheitswerte `true` und `false`:

```
jshell> x < x2
$18 ==> true
```

Wir können auch auf Wahrheitswerten rechnen. Hierzu gibt es drei logische Operationen. Das logische Oder, Disjunktion genannt:

```
jshell> $18 || $17
$19 ==> true
```

Das logische Und, Konjunktion genannt:

```
jshell> $18 && $17
$20 ==> false
```

Und die logische Nicht, Negation genannt:

```
jshell> !$20
$21 ==> true
```

Schließlich kann man auf Grund eines Wahrheitswertes eine Auswahl aus zwei Werten treffen.

```
jshell> $18?900:-42
$22 ==> 900
```

Da der Wert in der Variablen `$18` hier `true` ist, ist das Ergebnis `900`.

```
jshell> $17?900:-42
$23 ==> -42
```

Da der Wert in der Variablen `$18` hier `false` ist, ist das Ergebnis `-42`.

1.4.4 Ausdrücke für Zeichenketten

Häufig bearbeiten Computerprogramme Texte, die aus Zeichen eines Alphabets bestehen. Solche Zeichenketten lassen sich in Anführungszeichen angeben.

```
jshell> "Es ist noch etwas zu retten"
$15 ==> "Es ist noch etwas zu retten"
```

Der Datentyp, den Java hierzu verwendet heißt: `String`.

```
jshell> /vars
|    String $15 = "Es ist noch etwas zu retten"
```

Auch auf Daten des Typs `String` können wir in Java die Operation `+` durchführen. Dann wird eine neue Zeichenkette errechnet, die die beiden Teilketten aneinanderhängt.

```
jshell> $1+"! Handelt jetzt"
$16 ==> "Es ist noch etwas zu retten! Handelt jetzt"
```

Wir können auch eine Zahl mit dieser Operation an einen String anhängen. Dann wird die Zahl textuell als String umgewandelt und anschließend an den anderen String gehängt.

```
jshell> "Das Ergebnis von 17+4*2 ist: "+(17+4*2)
$17 ==> "Das Ergebnis von 17+4*2 ist: 25"
```

Ein wenig Vorsicht muss man dabei walten lassen. Betrachten Sie das folgende Beispiel.

```
jshell> "Das Ergebnis von 17+4*2 ist: "+17+4*2
$18 ==> "Das Ergebnis von 17+4*2 ist: 178"
```

Sehen Sie, was hier passiert ist?

Anders als einfache Zahlentypen, können wir Sringdaten nach ein paar Eigenschaften fragen. Zum Beispiel einen String nach der Anzahl der darin enthaltenen Buchstaben:

```
jshell> $17.length()
$19 ==> 31
```

Oder man kann für einen String einen neuen String errechnen lassen, der nur noch aus den entsprechenden Großbuchstaben besteht.

```
jshell> $17.toUpperCase()
$20 ==> "DAS ERGEBNIS VON 17+4*2 IST: 25"
```

Man kann nach dem Zeichen an einer bestimmten Position in einem String fragen:

```
jshell> $17.charAt(4)
$22 ==> 'E'
```

Das Ergebnis ist ein Wert des Datentyps `char`.

```
jshell> /var $22
|    char $22 = 'E'
```

Man kann auch einen neuen String errechnen lassen, indem bestimmte Teile durch andere Teilstrings ersetzt werden.

```
jshell> $17.replaceAll("i","IIIII")
$21 ==> "Das ErgebnIIIIIs von 17+4*2 IIIIIst: 25"
```

Man kann explizit einen Teilstring errechnen lassn.

```
jshell> $17.substring(4,10)
$23 ==> "Ergebn"
```

Es lässt sich allerhand für einen String errechnen. Wenn Sie in der jshell nach dem Punktsymbol die Tabulaturtaste drücken, dann zeigt Ihnen die jshell an, was Sie alles aus dem String errechnen lassen können.

```
jshell> $17.
charAt(              chars()            codePointAt(     codePointBefore(
codePointCount(      codePoints()       compareTo(       compareToIgnoreCase(
concat(              contains(          contentEquals(   endsWith(
equals(              equalsIgnoreCase(  getBytes(        getChars(
getClass()           hashCode()         indexOf(         intern()
isBlank()            isEmpty()          lastIndexOf(     length()
lines()              matches(           notify()         notifyAll()
offsetByCodePoints(  regionMatches(     repeat(          replace(
replaceAll(          replaceFirst(      split(           startsWith(
strip()              stripLeading()     stripTrailing()  subSequence(
substring(           toCharArray()      toLowerCase(     toString()
toUpperCase(         trim()             wait(

jshell>
```

1.4.5 Funktionen

Eine der wichtigsten Definitionen in der Mathematik dürfte die Definition des Begriffs Funktion sein. Viele Zweige der Mathematik beschäftigen sich mit nichts anderem als Funktionen und auch Sie haben in Ihrem Curriculum ein Modul, dass sich in der Analysis mit Funktionen beschäftigt. Aus der Schule sollte jeder wissen, was eine Funktion ist. Zum Beispiel die Funktion, die für jeder Zahl, das Quadrat der Zahl berechnet: $f(x) = x^2$

Wie wissen schon einmal, dass wir das Quadrat einer Zahl in Java durch die Multiplikation errechnen können.

```
jshell> 1*1
$1 ==> 1

jshell> 2*2
$2 ==> 4

jshell> 3*3
$3 ==> 9
```

Wir können eine Variable x zu Hilfe nehmen, in der wir erst den Wert speichern, der quadriert werden soll und das Quadrat der Variable x errechnen:

```
jshell> var x=1
x ==> 1

jshell> x*x
$4 ==> 1

jshell> x=2
x ==> 2

jshell> x*x
$5 ==> 4

jshell> x=3
x ==> 3

jshell> x*x
$6 ==> 9
```

Aber schließlich können wir auch wie in der Mathematik die Quadratfunktion defiineren

```
jshell> int f(int x){return x*x;}
|  created method f(int)
```

Jetzt gibt es die Funktion mit Namen `f` [3] und diese kann zum Rechnen für bestimmte Werte verwendet werden:

[3] Java sagt nicht Funktion sondern Methode, doch dazu später.

```
jshell> f(1)
$7 ==> 1

jshell> f(2)
$8 ==> 4

jshell> f(3)
$9 ==> 9

jshell> f(-17)
$10 ==> 289

jshell> f(17+4*2)
$11 ==> 625
```

In der Funktionsdefinition haben wir angegeben, von welchem Typ der Parameter x sein soll (`int x`) und von welchem Typ das Ergebnis wein soll (`int f`). Beides soll eine ganze Zahl sein. Somit ist auch das Ergebnis der Rechnung immer eine ganze Zahl:

```
jshell> /var $11
|    int $11 = 625
```

Die statische Typüberprüfung von Java verhindert, dass wir die Funktion für eine Kommazahl anwenden:

```
jshell> f(1.23)
|  Error:
|  incompatible types: possible lossy conversion from double to int
|  f(1.23)
|    ^--^
```

2 Grundkonzepte der Objektorientierung

Die Grundidee der objektorientierten Programmierung ist, Daten, die zusammen ein größeres zusammenhängendes Objekt beschreiben, zusammenzufassen. Zusätzlich fassen wir mit diesen Daten noch die Programmteile zusammen, die diese Daten manipulieren. Ein Objekt enthält also nicht nur die reinen Daten, die es repräsentiert, sondern auch Programmteile, die Operationen auf diesen Daten durchführen. Insofern wäre vielleicht *subjektorientierte Programmierung* ein passenderer Ausdruck, denn die Objekte sind nicht passive Daten, die von außen manipuliert werden, sondern enthalten selbst als integralen Bestandteil Methoden, die ihre Daten manipulieren können.

2.1 Interaktive Entwicklung von Klassen

Sie erinnern sich an den Java-Interpreter `jshell`? Bevor wir systematisch uns über Objekte, Klassen und deren Modellierung Gedanken machen, wollen wir diesen etwas spielerischen Ansatz nutzen, um eine erste Klasse zu entwickeln.

```
panitz@panitz-ThinkPad-T430:~/gitlab/LectureNotes$ jshell
|  Welcome to JShell -- Version 11.0.4
|  For an introduction type: /help intro
```

Als kleine Beispielsession soll eine Klasse, die Wetterdaten speichert entwickelt werden. Wir beginnen mit einer leeren Klasse:

```
jshell> class Wetter{}
|  created class Wetter

jshell> new Wetter()
$2 ==> Wetter@799f7e29
```

Nun soll diese Klasse ein Feld haben, in dem eine Temperatur in Grad Celsius gespeichert ist. Statt die neue Version der Klasse in der jshell neu einzugeben, können Sie: `/edit Wetter` eingeben. Dann geht der Standardeditor mit dem Quelltext der Klasse `Wetter` auf. Diesen können Sie bearbeiten. Speichern Sie diesen dann, so wird die Klasse mit dieser neuen Version in der jshell geladen.

```
jshell> class Wetter{
   ...>   int temperatur;
   ...> }
|  replaced class Wetter

jshell> new Wetter()
$4 ==> Wetter@7aec35a
```

Jetzt können wir dem Wetterobjekt einen konkreten Wert im Feld `temperatur` zuweisen.

```
jshell> $4.temperatur=21
$5 ==> 21
```

Das in `$4` gespeicherte Wetterobjekt hat jetzt den zuvor zugewiesenen Wert.

```
jshell> $4.temperatur
$6 ==> 21
```

Als nächsten Schritt bekommt die Klasse einen Konstruktor. Zum Erzeugen eines Wetterobjektes ist jetzt ein konkreter Wert für die Temperatur im Konstruktor zu übergeben.

```
jshell> class Wetter{
   ...>   int temperatur;
   ...>   Wetter(int t){
   ...>     temperatur = t;
   ...>   }
   ...> }
|  replaced class Wetter

jshell> var w = new Wetter(21);
w ==> Wetter@1de0aca6
```

Das in `w` gespeicherte Wetterobjekt hat jetzt den im Konstruktor übergebenen Wert.

```
jshell> w.temperatur
$9 ==> 21
```

Als nächstes wird die Klasse um eine Methode erweitert, die die gespeicherte Temperatur in Fahrenheit angibt.

```
jshell> class Wetter{
   ...>   int temperatur;
   ...>   Wetter(int t){
   ...>     temperatur = t;
   ...>   }
   ...>   double temperaturInFahrenheit(){
   ...>     return temperatur * 1.8 + 32;
   ...>   }
   ...> }
|  replaced class Wetter
|    update replaced variable w, reset to null

jshell> var w = new Wetter(21);
w ==> Wetter@30946e09
```

Jetzt können wir ein Wetterobjekt mit einem Methodenaufruf nach der Temperatur in Fahrenheit fragen:

```
jshell> w.temperaturInFahrenheit()
$12 ==> 69.80000000000001
```

natürlich steht weiterhin auch das Feld, in dem die Temperatur in Celsius gespeichert ist, zur Verfügung.

```
jshell> w.temperatur
$13 ==> 21
```

Schließlich soll unsere Klasse noch eine Methode `toString` enthalten. In dieser soll für die textuelle Darstellung eines Wetterobjektes sowohl die Temperatur in Celsius als auch in Fahrenheit verwendet werden. Hierzu wird innerhalb der Methode `toString` sowohl auf das Feld `temperatur` als auch die Methode `temperaturInFahrenheit()` zugegriffen.

```
jshell> class Wetter{
   ...>   int temperatur;
   ...>   Wetter(int t){
   ...>     temperatur = t;
   ...>   }
   ...>   double temperaturInFahrenheit(){
   ...>     return temperatur * 1.8 + 32;
   ...>   }
   ...>   public String toString(){
   ...>     return "Es sind "+temperatur+" Grad Celsius, "
   ...>       +"das entspricht "+temperaturInFahrenheit()
   ...>       +" Grad Fahrenheit.";
   ...>   }
   ...> }
|  replaced class Wetter
|    update replaced variable w, reset to null
```

Die jshell verwendet jetzt automatisch die `toString`-Methode bei der Anzeige eines Wetterobjektes.

```
jshell> var w = new Wetter(21);
w ==> Es sind 21 Grad Celsius, das entspricht 69.80000000000001 Grad Fahrenheit.
```

Am Ende einer solchen interaktiven Session zur Entwicklung einer Klasse, können wir diese mit der jshell-Anweisung `/save` in eine Javaquelltextdatei abspeichern.

```
jshell> /save Wetter Wetter.java
```

Wem das zu schnell war, für den gehen wir in den nächsten Abschnitten der Entwicklung von Klassen etwas systematischer nach.

2.2 Objektorientierte Modellierung

Jetzt wollen wir erst einmal eine informelle Modellierung der Welt, für die ein Programm geschrieben werden soll, vornehmen. Hierzu empfiehlt es sich durchaus, in einem Team zusammenzusitzen und auf Karteikarten aufzuschreiben, was es denn für Objekte in der Welt gibt, die wir modellieren wollen.

Stellen wir uns hierzu einmal vor, wir sollen ein Programm zur Bibliotheksverwaltung schreiben. Jetzt überlegen wir einmal, was gibt es denn für Objektarten, die alle zu den Vorgängen in einer Bibliothek gehören. Hierzu fällt uns vielleicht folgende Liste ein:

- Personen, die Bücher ausleihen wollen.
- Bücher, die ausgeliehen werden können.

- Tatsächliche Ausleihvorgänge, die ausdrücken, dass ein Buch bis zu einem bestimmten Zeitpunkt von jemanden ausgeliehen wurde.
- Termine, also Objekte, die ein bestimmtes Datum kennzeichnen.

Nachdem wir uns auf diese vier für unsere Anwendung wichtigen Objektarten geeinigt haben, nehmen wir vier Karteikarten und schreiben jeweils eine der Objektarten als Überschrift auf diese Karteikarten.

Jetzt haben wir also Objektarten identifiziert. Im nächsten Schritt ist zu überlegen, was für Eigenschaften diese Objekte haben. Beginnen wir für die Karteikarte, auf der wir als Überschrift *Person* geschrieben haben. Was interessiert uns an Eigenschaften einer Person? Wahrscheinlich ihr Name mit Vorname, Straße und Ort sowie Postleitzahl. Das sollten die Eigenschaften einer Person sein, die für ein Bibliotheksprogramm notwendig sind. Andere mögliche Eigenschaften wie Geschlecht, Alter, Beruf oder ähnliches interessieren uns in diesem Kontext nicht. Jetzt schreiben wir die Eigenschaften, die uns von einer Person interessieren, auf die Karteikarte mit der Überschrift Person.

Schließlich müssen wir uns Gedanken darüber machen, was diese Eigenschaften eigentlich für Daten sind. Name, Vorname, Straße und Wohnort sind sicherlich als Texte abzuspeichern oder, wie der Informatiker gerne sagt, als Zeichenketten. Die Postleitzahl ist hingegen als eine Zahl abzuspeichern. Diese Art, von der die einzelnen Eigenschaften sind, nennen wir ihren Typ. Wir schreiben auf die Karteikarte für die Objektart *Person* hinter jede der Eigenschaften noch den Typ, den diese Eigenschaft hat. Damit erhalten wir für die Objektart *Person* die in Abbildung 2.1 gezeigte Karteikarte.

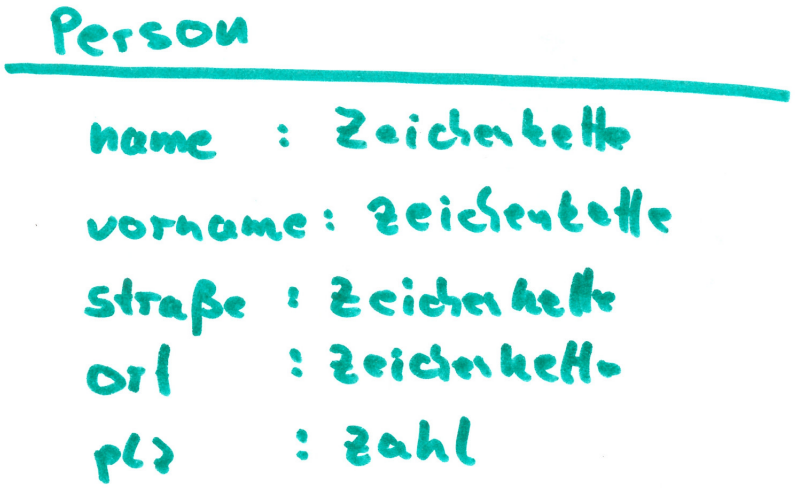

Abbildung 2.1: Modellierung einer Person.

Gleiches können wir für die Objektart *Buch* und für die Objektart *Datum* machen. Wir

erhalten dann die Karteikarten aus Abbildung 2.2 und 2.3 .

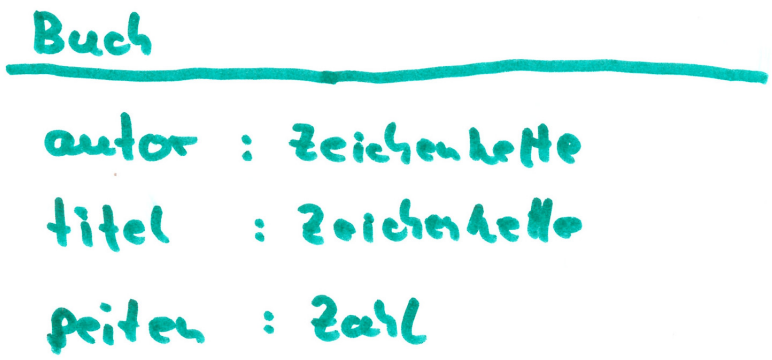

Abbildung 2.2: Modellierung eines Buches.

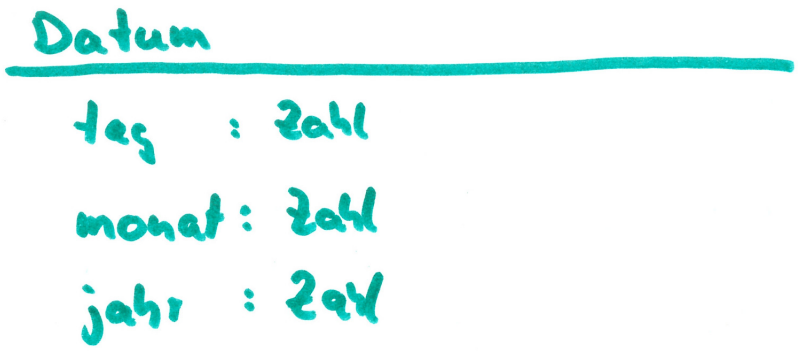

Abbildung 2.3: Modellierung eines Datums.

Wir müssen uns schließlich nur noch um die Objektart einer Buchausleihe kümmern. Hier sind drei Eigenschaften interessant: wer hat das Buch geliehen, welches Buch wurde verliehen und wann muss es zurückgegeben werden. Wir können also drei Eigenschaften auf die Karteikarte schreiben. Was sind die Typen dieser drei Eigenschaften? Diesmal sind es keine Zahlen oder Zeichenketten, sondern Objekte der anderen drei bereits modellierten Objektarten. Wenn wir nämlich eine Karteikarte schreiben, dann erfinden wir gerade einen neuen Typ, den wir für die Eigenschaften anderer Karteikarten benutzen können.

Somit erstellen wir eine Karteikarte für den Objekttyp *Ausleihe*, wie sie in Abbildung 2.4 zu sehen ist.

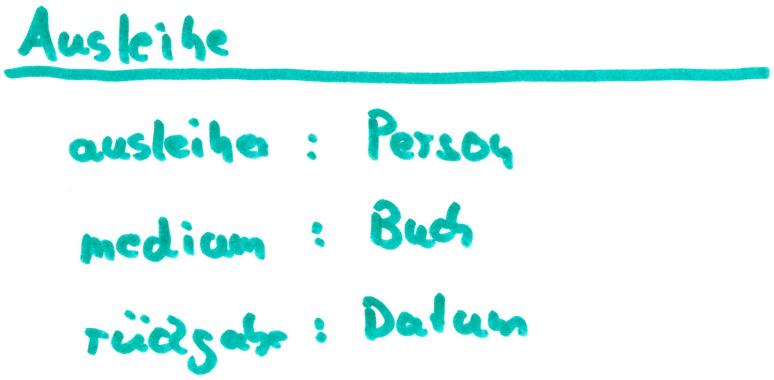

Abbildung 2.4: Modellierung eines Ausleihvorgangs.

2.3 Klassen und Objekte

Wir haben in einem Modellierungsschritt im letzten Abschnitt verschiedene Objektarten identifiziert und ihre Eigenschaften spezifiziert. Dazu haben wir vier Karteikarten geschrieben. Jetzt können wir versuchen, diese Modellierung in Java umzusetzen. In Java beschreibt eine Klasse eine Menge von Objekten gleicher Art. Damit entspricht eine Klasse einer der Karteikarten in unserer Modellierung. Die Klassendefinition ist eine Beschreibung der möglichen Objekte. In ihr ist definiert, was für Daten zu den Objekten gehören. Zusätzlich können wir in einer Klasse noch schreiben, welche Operationen auf diesen Daten angewendet werden können. Klassendefinitionen sind die eigentlichen Programmtexte, die der Programmierer schreibt.

Merken Sie sich:

In Java steht genau eine Klassendefinition[a] in genau einer Datei. Die Datei hat dabei den Namen der Klasse mit der Endung `.java`.

[a] Auch hier werden wir Ausnahmen kennenlernen.

Jede Programmiersprache hat ein paar Wörter, die eine für die Sprache besondere Bedeutung haben. Diese Wörter sind festgelegt und werden als Schlüsselwörter bezeichnet. Die Zahl der Schlüsselwörter ist meist recht klein gehalten.

Zu Beginn einer Klassendefinition steht in Java das Schlüsselwort `class`, gefolgt von dem Namen, den man für die Klasse gewählt hat. Der Name der Klasse ist frei wählbar. Wann immer der Programmierer in einer Programmiersprache einen Namen, sei es für eine Klasse, eine Variable oder auch für Felder und Methoden, frei wählen kann, nennt man diese Namen auch *Bezeichner* (engl. *identifier*).

Nach dem Bezeichner für den Klassennamen folgt in geschweiften Klammern der Inhalt der Klasse bestehend aus Felddefinitionen und Methodendefinitionen.

Die einfachste Klasse, die in Java denkbar ist, ist eine Klasse ohne Felder oder Methoden:

Minimal.java

```
class Minimal {
}
```

Merken Sie sich:

Groß- und Kleinschreibung ist in Java relevant. Alle Schlüsselwörter wie `class` werden stets klein geschrieben. Klassennamen starten per Konvention immer mit einem Großbuchstaben.

2.3.1 Felder

Zum Speichern von Daten können Felder (engl. *field*) für eine Klasse definiert werden. In einem Feld können Objekte für eine bestimmte Klasse gespeichert werden. Bei der Felddeklaration wird angegeben, welche Art von Objekten in einem Feld abgespeichert werden sollen. Die Felder entsprechen dabei genau den Eigenschaften, die wir auf unsere Karteikarten geschrieben haben.

Syntaktisch wird in Java der Klassenname des Typs, von dem Objekte gespeichert werden sollen, den frei zu wählenden Feldnamen vorangestellt. Eine Felddeklaration endet mit einem Semikolon.

Im Folgenden schreiben wir eine Klasse mit drei Feldern:

Buch.java

```
class Buch{
  String autor;
  String titel;
  int preisInCent;
}
```

Für eine Zeichenkette steht in Java die Klasse `String` zur Verfügung. Für ganze Zahlen existiert der Typ `int`.

Die Reihenfolge in der die Felder einer Klasse definiert werden, ist mehr oder weniger irrelevant.

In der Literatur werden Felder auf deutsch auch als Attribute einer Klasse oder auch als Exemplarvariablen bezeichent.

2.3.2 Objekte

Im letzten Abschnitt haben wir eine erste Klasse geschrieben. Sie implementiert die Modellierung von Büchern. Sie hat daher drei Felder, die den Eigenschaften aus der Modellierung entsprechen. Jetzt wollen wir diese Klasse benutzen, um damit konkrete Buchobjekte zu erzeugen, die konkrete Werte für den Titel etc. haben. Hierzu brauchen wir eine Klasse mit einer Hauptmethode, die ausgeführt werden soll. Innerhalb dieser Hauptmethode soll dann ein Buchobjekt erzeugt werden.

Wir beginnen eine entsprechende Klasse zum ersten Testen von Buchobjekten.

TestBuch.java

```
class TestBuch{
  public static void main(String[] args){
```

Erzeugen von Objekten mit `new`

Innerhalb dieser Hauptmethoden können wir jetzt konkrete Objekte der Klasse `Buch` erzeugen. Hierzu gibt es in Java das Schlüsselwort `new`. Mit diesem wird angezeigt, dass ein neues Objekt erzeugt werden soll. Dem Schlüsselwort folgt der Name der Klasse, für die ein neues Objekt zu erzeugen ist. In unserem Fall also `Buch`. Anschließend folgt noch ein rundes Klammernpaar, also insgesamt `new Buch()`.

Wurde so ein Objekt erzeugt, ist dieses irgendwo abzuspeichern. Hierzu dienen lokale Variablen. Diese haben einen Bezeichner. Über diesen Bezeichner wird die Variable angesprochen. Zusätzlich ist noch durch einen Typnamen anzugeben, welche Art von Daten in dieser Variable gespeichert werden sollen. In unserem Fall ist dieses der Typ `Buch`, denn es sollen Buchobjekte in der Variablen abgelegt werden.

TestBuch.java

```
    Buch b1 = new Buch();
```

Das Gleichheitszeichen in dieser Zeile steht für die sogenannte *Zuweisung* (eng.: *assignment*). Der Variablen `b1` wird hiermit ein konkretes Objekt zugewiesen.

Eine Variable hat in Java immer einen festen Typ, der festlegt, welche Art von Daten in der Variable gespeichert werden sollen. Dieser Typ wird bei der Variablendeklaration den Namen der Variable vorangestellt. In diesem Beispiel ist es der Typ der Klasse `Buch`. Bei lokalen Variablen kann man statt des festen Typs auch das Schlüsselwort `var` verwenden.

TestBuch.java

```
    var b2 = new Buch();
```

Dieses bedeutet nicht, dass die Variable keinen festen Typ hat. Der Kompilator errechnet nur diesen Typ aus den Kontext und der Programmierer brauch den Typ somit nicht explizit anzugeben. trotzdem hat in diesem Beispiel die Variable `b2` auch den festen Typ `Buch`.

Zugriff auf die Felder von Objekten

Von nun an bezeichnet die Variable `b1` ein konkretes Objekt der Klasse `Buch`. Allerdings haben wir für dieses Objekt noch keine konkreten Werte für die Felder zugewiesen. Der Titel, der Autor und auch der Preis für dieses Buchobjekt wurde noch nicht gesetzt. Auf die Felder eines Objektes kann mit einem Punkt zugegriffen werden. Links von dem Punkt steht ein Objekt, rechts davon ein Feldname. So können wir in unserem Beispiel mit `b1.titel` auf den Titel des Buchobjektes `b1` zugreifen. Um einen Titel zu setzen, kann wieder der Zuweisungsbefehl genutzt werden. Wir können alle drei Felder mit folgenden Zuweisungen mit konkreten Werten belegen:

TestBuch.java

```
b1.titel = "Geschüttelt, nicht gerührt!";
b1.autor = "Matthias Oheim, Sven Eric Panitz";
b1.preisInCent = 750;
```

Wie man sieht, lassen sich Zeichenketten in Java direkt in doppelten Anführungszeichen eingeschlossen hinschreiben. Dabei ist zu beachten, dass diese so notierten Stringobjekte nicht über ein Zeilenende hinaus gehen dürfen.

Konstante Zahlen lassen sich in Java wie eigentlich in allen Programmiersprachen direkt hinschreiben.

Nach den obigen Zuweisungen hat das Objekt `b1` konkrete Werte. Mit dem Ausgabebefehl, den wir im vorherigen Kapitel vorgestellt haben, lassen sich die einzelnen Daten der Felder eines Objektes auf der Kommandozeile ausgeben:

TestBuch.java

```
System.out.println(b1.autor);
```

Klassen werden geschrieben, um nicht nur ein Objekt, sondern zumeist sehr viele Objekte einer Klasse zu erzeugen. Somit können wir jetzt ein zweites Buchobjekt erzeugen und diesem konkrete Werte für die Felder zuweisen.

TestBuch.java

```
Buch b2 = new Buch();
b2.titel = "Gerüttelt, nicht geschürt!";
b2.autor = "Matthias Oheim, Sven Eric Panitz";
b2.preisInCent = 980;
```

Die Objekte einer Klasse sind voneinander unabhängig. Beide Objekte haben ihre eigenen Werte. Davon können wir uns durch die Ausgabe der Titel beider Objekte überzeugen.

TestBuch.java

```
    System.out.println(b1.titel);
    System.out.println(b2.titel);
```

Soweit unsere erste Klasse, die mit Buchobjekten arbeitet. Es fehlen noch die zwei schließenden geschweiften Klammern. Einmal, um anzuzeigen, dass die Hauptmethode beendet wird und einmal um das Ende des Klassendefinition anzuzeigen.

TestBuch.java

```
  }
}
```

Merken Sie sich:

Java benutzt Paare von geschweiften Klammern um Blöcke mit Anfang und Ende zu markieren. Per Konvention wird ein solch geklammerter Block eine Ebene weiter eingerückt als der Kontext.

Konstruktoren

Wir haben oben gesehen, wie prinzipiell Objekte einer Klasse mit dem `new`-Konstrukt erzeugt werden. In unserem obigen Beispiel würden wir gerne bei der Erzeugung eines Objektes gleich konkrete Werte für die Felder mit angeben, um direkt ein Buch mit konkreten Titel, Autor und Preis erzeugen zu können. Hierzu können Konstruktoren für eine Klasse definiert werden.

Machen wir dieses exemplarisch für die Klasse, die Personen modelliert.

Person.java

```
class Person {
  String vorname;
  String nachname;

  Person(String derVorname, String derNachname){
    vorname  = derVorname;
    nachname = derNachname;
  }
}
```

Jetzt lassen sich bei der Erzeugung von Objekten des Typs `Person` konkrete Werte für die Namen übergeben.

Wir erzeugen ein Personenobjekt mit dem für die entsprechende Klasse geschriebenen Konstruktor:

TestePerson1.java

```
class TestePerson {

  public static void main(String [] args){
    Person p = new Person("Nicolo","Paganini");
    System.out.println(p.vorname);
  }
}
```

2.3.3 Objekte der Klasse `String`

Java kommt bereits mit einer großen Anzahl zur Verfügung stehender Standardklassen. Es müssen also nicht alle Klassen neu vom Programmierer definiert werden. Eine sehr häufig benutzte Klasse ist die Klasse `String`. Sie repräsentiert Objekte, die eine Zeichenkette darstellen, also einen Text, wie wir ihn in unserer ersten Modellierung bereits vorausgesetzt haben.

Für die Klasse `String` gibt es eine besondere Art, Objekte zu erzeugen. Ein in Anführungsstrichen eingeschlossener Text erzeugt ein Objekt der Klasse `String`.

Aus zwei Objekten der Stringklasse läßt sich ein neues Objekt erzeugen, indem diese beiden Objekte mit einem Pluszeichen verbunden werden:

```
"hallo "+"welt"
```

Hier werden die zwei Stringobjekte `"hallo "` und `"welt"` zum neuen Objekt `"hallo welt"` verknüpft.

Der Plus-Operator hat für Objekte der Stringklasse noch eine besondere Bedeutung und stellt damit eine Besonderheit innerhalb der Programmiersprache Java dar.

Erzeugen wir zwei String-Objekte:

```
jshell> String s1 = "hallo";
s1 ==> "hallo"

jshell> var s2 = "welt";
s2 ==> "welt"
```

Und legen eine int-variable an:

```
jshell> int i = 42;
i ==> 42
```

Wir können mit dem +-Operator beliebig Strings und andere Daten zu einen neuen String aneinanderhängen.

```
jshell> String s3 = s1+" "+s2+": "+42;
s3 ==> "hallo welt: 42"
```

Dadurch werden keine bestehenden String-Objekte verändert.

```
jshell> s1
s1 ==> "hallo"
```

Man sollte nur bei heterogenen Daten in dieser Operation ein wenig vorsichtig sein.

```
jshell> ""+i+1
$49 ==> "421"

jshell> ""+(i+1)
$50 ==> "43"
```

Wie kommt es zu den unterschiedlichen Ergebnissen?

Merken Sie sich:

Auch `String` ist eine Klasse, genauso wie alle anderen Klassen. Der einzige Unterschied ist, dass es die Möglichkeit gibt, String-Objekte durch in Anführungszeichen eingeschlossenen Text zu erzeugen, statt durch ein Aufruf eines Konstruktors mit `new`.

2.3.4 Methoden

Methoden [1] sind die Programmteile, die in einer Klasse definiert sind und für jedes Objekt dieser Klasse zur Verfügung stehen. Die Ausführung einer Methode liefert meist ein Ergebnisobjekt. Methoden haben eine Liste von Eingabeparametern. Ein Eingabeparameter ist durch den gewünschten Klassennamen und einen frei wählbaren Parameternamen spezifiziert.

Methodendeklaration

In Java wird eine Methode deklariert durch: den Rückgabetyp, den Namen der Methode, der in Klammern eingeschlossenen durch Kommas getrennten Parameterliste und den

[1] Der Ausdruck für *Methoden* kommt speziell aus der objektorientierten Programmierung. In der imperativen Programmierung spricht man von *Prozeduren*, die funktionale Programmierung von *Funktionen*. Weitere Begriffe, die Ähnliches beschreiben, sind *Unterprogramme* und *Subroutinen*.

in geschweiften Klammern eingeschlossenen Programmrumpf. Im Programmrumpf wird mit dem Schlüsselwort `return` angegeben, welches Ergebnisobjekt die Methode liefert.

Als Beispiel definieren wir eine Klasse, in der es eine Methode `addString` gibt, die den Ergebnistyp `String` und zwei Parameter vom Typ `String` hat:

StringUtilMethod.java

```
class StringUtilMethod {
  String addStrings(String leftText, String rightText){
    return leftText+rightText;
  }
}
```

Merken Sie sich:

Methoden und Parameternamen werden per Konvention immer klein geschrieben. Besteht der Bezeichnername aus mehreren Wörtern, so wird innerhalb des Bezeichners jedes Wort mit einem Großbuchstaben begonnen. Dieses bezeichnet man als *camel case* Notation.

Zugriff auf Felder im Methodenrumpf

In einer Methode stehen die Felder der Klasse zur Verfügung[2].

Wir können mit den bisherigen Mitteln eine kleine Klasse definieren, die es erlaubt, Personen zu repräsentieren, so dass die Objekte dieser Klasse eine Methode haben, um den vollen Namen der Person anzugeben:

PersonExample1.java

```
class PersonExample1 {
  String vorname;
  String nachname;

  String getFullName(){
    return (vorname+" "+nachname);
  }
}
```

Methoden ohne Rückgabewert

Es lassen sich auch Methoden schreiben, die keinen eigentlichen Wert berechnen, den sie als Ergebnis zurückgeben. Solche Methoden haben keinen Rückgabetyp. In Java wird

[2]Das ist wiederum nicht die volle Wahrheit, wie in Kürze zu sehen sein wird.

dieses gekennzeichnet, indem das Schlüsselwort `void` statt eines Typnamens in der Deklaration steht. Solche Methoden haben keine `return`-Anweisung.

Folgende kleine Beispielklasse enthält zwei Methoden zum Setzen neuer Werte für ihre Felder:

PersonExample2.java

```
class PersonExample2 {
  String vorname;
  String nachname;

  void setVorname(String newName){
    vorname = newName;
  }
  void setNachname(String newName){
    nachname = newName;
  }
}
```

Obige Methoden weisen konkrete Objekte den Feldern des Objektes zu.

Merken Sie sich:

Eine Klasse in Java hat Felder, Methoden und Konstruktoren.

2.4 Der Bezeichner `this`

Eigenschaften sind an Objekte gebunden. Es gibt in Java eine Möglichkeit, in Methodenrümpfen über das Objekt, für das eine Methode aufgerufen wurde, zu sprechen. Dieses geschieht mit dem Schlüsselwort `this`. *this* ist zu lesen als: dieses Objekt, in dem du dich gerade befindest. Häufig wird der `this`-Bezeichner in Konstruktoren benutzt, um den Namen des Parameters des Konstruktors von einem Feld zu unterscheiden:

UseThis.java

```
class UseThis {
  String aField ;
  UseThis(String aField){
    this.aField = aField;
  }
}
```

2.5 Statische Eigenschaften einer Klasse

Bisher haben wir Methoden und Felder kennengelernt, die immer nur als Teil eines konkreten Objektes existiert haben. Es gibt aber auch Felder und Methoden, die unabhängig von konkreten Objekten für eine Klasse existieren

2.5.1 Statische Methoden

Es muss auch eine Möglichkeit geben, Methoden, die unabhängig von Objekten existieren, zu deklarieren, weil wir ja mit irgendeiner Methoden anfangen müssen, in der erst Objekte erzeugt werden können. Hierzu gibt es statische Methoden. Die Methoden, die immer an ein Objekt gebunden sind, heißen im Gegensatz dazu dynamische Methoden. Statische Methoden brauchen kein Objekt, um aufgerufen zu werden. Sie werden exakt so

deklariert wie dynamische Methoden, mit dem einzigen Unterschied, dass ihnen das Schlüsselwort `static` vorangestellt wird. Statische Methoden werden auch in einer Klasse definiert und gehören zu einer Klasse. Da statische Methoden nicht zu den einzelnen Objekten gehören, können sie nicht auf dynamische Felder und Methoden der Objekte zugreifen.

StaticTest.java

```
class StaticTest {
  static void printThisText(String text){
    System.out.println(text);
  }
}
```

Statische Methoden werden direkt auf der Klasse, nicht auf einem Objekt der Klasse aufgerufen.

Auf statische Eigenschaften wird zugegriffen, indem vom Klassennamen per Punkt getrennt die Eigenschaft aufgerufen wird:

CallStaticTest.java

```
class CallStaticTest {
  public static void main(String [] args){
    StaticTest.printThisText("hello");
  }
}
```

Es gibt die Standardklasse `Math` in der mathematische Konstanten und Standardfunktionen als statische Eigenschaften implementiert sind. Hier finden sich z.B. die trigonomischen Funktionen und vieles mehr:

```
jshell> Math.pow(17,3)
$21 ==> 4913.0

jshell> Math.cos(Math.PI)
$22 ==> -1.0

jshell> Math.sin(Math.PI)
$23 ==> 1.2246467991473532E-16

jshell> Math.atan(1.65657)
$24 ==> 1.0276922275770104

jshell> Math.max(17,4)
$25 ==> 17
```

2.5.2 Statische Felder

Ebenso wie statische Methoden gibt es auch statische Felder. Im Unterschied zu dynamischen Feldern existieren statische Felder genau einmal, nämlich in der Klasse. Dynamische Felder existieren für jedes Objekt der Klasse.

Hierzu betrachte man das einfache Beispiel einer Klasse mit einem statischen Feld:

```
jshell> class Person{
   ...>   static long totalNumber = 0;
   ...>   String name;
   ...>   Person(String n){
   ...>     name = n;
   ...>     totalNumber = totalNumber + 1;
   ...>   }
   ...> }
|  created class Person
```

Von dieser lässt sich ein Objekt über den Konstruktor erzeugen.

```
jshell> new Person("Jupp")
$10 ==> Person@5a2e4553
```

Im Konstruktor wird das statische Feld `totalNumber`, das nur einmal für alle Objekte existiert, um eins erhöht.

```
jshell> Person.totalNumber
$11 ==> 1
```

Das sieht man daran, dass nach der Erzeugung eines weiteren Objektes, dieses Feld dannm den Wert 2 hat.

```
jshell> new Person("Egon")
$12 ==> Person@6d5380c2

jshell> Person.totalNumber
$13 ==> 2
```

Man kann statische Felder auch über die Objekte einer Klasse ansprechen. Es wird aber empfohlen dieses nicht zu tun, weil sonst der Eindruck entsteht, dass jedes Objekt eigene Version dieses Feldes hat.

```
jshell> $12.totalNumber
$14 ==> 2

jshell> $10.totalNumber
$15 ==> 2
```

Merken Sie sich:

Mit statischen Eigenschaften verlässt man quasi die objektorientierte Programmierung, denn diese Eigenschaften sind nicht mehr an Objekte gebunden. Statische Eigenschaften nennt man auch Klassenfelder bzw. Klassenmethoden.

3 Imperative und funktionale Konzepte

Im letzten Abschnitt wurde ein erster Einstieg in die objektorientierte Programmierung gegeben. Wie zu sehen war, ermöglicht die objektorientierte Programmierung, das zu lösende Problem in logische Untereinheiten zu unterteilen, die direkt mit den Teilen der zu modellierenden Problemwelt korrespondieren.

Die Methodenrümpfe, die die eigentlichen Befehle enthalten, in denen etwas berechnet werden soll, waren bisher recht kurz. In diesem Kapitel werden wir Konstrukte kennenlernen, die es ermöglichen, in den Methodenrümpfen komplexe Berechnungen vorzunehmen. Die in diesem Abschnitt vorgestellten Konstrukte sind herkömmliche Konstrukte der imperativen Programmierung und in ähnlicher Weise auch in Programmiersprachen wie C zu finden.[1] Wir können diese konstrukte alle gut mit Hilfe der jshell ausprobieren.

3.1 Primitive Typen

Bisher haben erst wenige Berechnungen im klassischen Sinne als das Rechnen mit Zahlen kennengelernt. Java stellt Typen zur Repräsentation von Zahlen zur Verfügung. Zwei davon haben wir schon gesehen: `int` und `double`. Leider sind diese Typen keine Klassen; d.h. insbesondere, dass auf diesen Typen keine Felder und Methoden existieren, auf die mit einem Punkt zugegriffen werden kann.

Die im Folgenden vorgestellten Typen nennt man primitive Typen. Sie sind fest von Java vorgegeben. Im Gegensatz zu Klassen, die der Programmierer selbst definieren kann, können keine neuen primitiven Typen definiert werden. Um primitive Typnamen von Klassennamen leicht textuell unterscheiden zu können, sind sie in Kleinschreibung definiert worden.

Ansonsten werden primitive Typen genauso behandelt wie Klassen. Felder können primitive Typen als Typ haben und ebenso können Parametertypen und Rückgabetypen von Methoden primitive Typen sein.

Um Daten der primitiven Typen aufschreiben zu können, gibt es jeweils Literale für die Werte dieser Typen.

[1] Gerade in diesem Bereich wollten die Entwickler von Java einen leichten Umstieg von der C-Programmierung nach Java ermöglichen. Leider hat Java in dieser Hinsicht auch ein C-Erbe und ist nicht in allen Punkte so sauber entworfen, wie es ohne diese Designvorgabe wäre.

3.1.1 Zahlenmengen in der Mathematik

In der Mathematik sind wir gewohnt, mit verschiedenen Mengen von Zahlen zu arbeiten:

- **natürliche Zahlen** $\mathbb{N}$: Eine induktiv definierbare Menge mit einer kleinsten Zahl, so dass es für jede Zahl eine eindeutige Nachfolgerzahl gibt.
- **ganze Zahlen** $\mathbb{Z}$: Die natürlichen Zahlen erweitert um die mit einem negativen Vorzeichen behafteten Zahlen, die sich ergeben, wenn man eine größere Zahl von einer natürlichen Zahl abzieht.
- **rationale Zahlen** $\mathbb{Q}$: Die ganzen Zahlen erweitert um Brüche, die sich ergeben, wenn man eine Zahl durch eine Zahl teilt, von der sie kein Vielfaches ist.
- **reelle Zahlen** $\mathbb{R}$: Die ganzen Zahlen erweitert um irrationale Zahlen, die sich z.B. aus der Quadratwurzel von Zahlen ergeben, die nicht das Quadrat einer rationalen Zahl sind.
- **komplexe Zahlen** $\mathbb{C}$: Die reellen Zahlen erweitert um imaginäre Zahlen, wie sie benötigt werden, um einen Wurzelwert für negative Zahlen darzustellen.

Es gilt folgende Mengeninklusion zwischen diesen Mengen:

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

Da bereits $\mathbb{N}$ nicht endlich ist, ist keine dieser Mengen endlich.

3.1.2 Zahlenmengen im Rechner

Da wir nur von einer endlich großen Speicherkapazität ausgehen können, lassen sich für keine der aus der Mathematik bekannten Zahlenmengen alle Werte in einem Rechner darstellen. Wir können also schon einmal nur Teilmengen der Zahlenmengen darstellen.

Von der Hardwareseite stellt sich heute zumeist die folgende Situation dar: Der Computer hat einen linearen Speicher, der in Speicheradressen unterteilt ist. Eine Speicheradresse bezeichnet einen Bereich von 32 Bit. Wir bezeichnen diese als ein Wort. Die Einheit von 8 Bit wird als Byte bezeichnet [2]. Heutige Rechner verwalten also in der Regel Dateneinheiten von 32 Bit. Hieraus ergibt sich die Kardinalität der Zahlenmengen, mit denen ein Rechner als primitive Typen rechnen kann. Soll mit größeren Zahlenmengen gerechnet werden, so muss hierzu eine Softwarelösung genutzt werden.

Ganzzahlentypen in Java

Java 4 Typen zur Darstellung ganzer Zahlen, die sich lediglich in der Anzahl der Ziffern unterscheiden. Die Zahlen werden intern als Zweierkomplement dargestellt.

[2] ein anderes selten gebrauchtes Wort aus dem Französischen ist: Oktett

Typname	Länge	Wertebereich
`byte`	8 Bit	$-128 = -2^7$ bis $127 = 2^7 - 1$
`short`	16 Bit	$-32768 = -2^{15}$ bis $32767 = 2^{15} - 1$
`int`	32 Bit	$-2147483648 = -2^{31}$ bis $2147483647 = 2^{32} - 1$
`long`	64 Bit	-9223372036854775808 bis 9223372036854775807

In der Programmiersprache Java sind die konkreten Wertebereiche für die einzelnen primitiven Typen in der Spezifikation festgelegt. In anderen Programmiersprachen wie z.B. C ist dies nicht der Fall. Hier hängt es vom Compiler und dem konkreten Rechner ab, welchen Wertebereich die entsprechenden Typen haben.

Es gibt Programmiersprachen wie z.B. Haskell, in denen es einen Typ gibt, der potentiell ganze Zahlen von beliebiger Größe darstellen kann.

Betrachten Sie die Auswertung folgender Javaausdrücke:

```
jshell> 2147483647+1
$27 ==> -2147483648

jshell> -2147483648-1
$28 ==> 2147483647
```

Erklären Sie die Ausgabe.

Fließkommazahlen in Java

Eine Alternative zu der Festkommadarstellung von Zahlen ist die Fließkommadarstellung. Während die Festkommadarstellung einen Zahlenbereich der rationalen Zahlen in einem festen Intervall durch diskrete, äquidistant verteilte Werte darstellen kann, sind die diskreten Werte in der Fließkommadarstellung nicht gleich verteilt.

In der Fließkommadarstellung wird eine Zahl durch zwei Zahlen charakterisiert und ist bezüglich einer Basis b:

- die Mantisse für die darstellbaren Ziffern. Die Mantisse charakterisiert die Genauigkeit der Fließkommazahl.
- der Exponent, der angibt, wie weit die Mantisse hinter bzw. vor dem Komma liegt.

Aus Mantisse m, Basis b und Exponent exp ergibt sich die dargestellte Zahl durch folgende Formel:

$$z = m * b^{exp}$$

Damit lassen sich mit Fließkommazahlen sehr große und sehr kleine Zahlen darstellen. Je größer jedoch die Zahlen werden, desto weiter liegen sie von der nächsten Zahl entfernt.

Für die Fließkommadarstellung gibt es in Java zwei Zahlentypen, die nach der Spezifikation des IEEE 754-1985 gebildet werden:

- **float**: 32 Bit Fließkommazahl nach IEEE 754. Kleinste positive Zahl: 2^{-149}. Größte positive Zahl: $(1 - 2^{-24}) * 2^{128}$

- **double**: 64 Bit Fließkommazahl nach IEEE 754. Kleinste positive Zahl: 2^{-1074}. Größte positive Zahl: $(1 - 2^{-53}) * 2^{1024}$

Im Format für `double` steht das erste Bit für das Vorzeichen, die nächsten 11 Bit markieren den Exponenten und die restlichen 52 Bit kodieren die Mantisse.

Im Format für `float` steht das erste Bit für das Vorzeichen, die nächsten 8 Bit markieren den Exponenten und die restlichen 23 Bit kodieren die Mantisse.

Bestimmte Bitmuster charakterisieren einen Wert für negative und positive unbeschränkte Werte (unendlich) sowie Zahlen, Bitmuster, die charakterisieren, dass es sich nicht mehr um eine Zahl handelt.

Der folgende Test zeigt, dass bei einer Addition von zwei Fließkommazahlen die kleinere Zahl das Nachsehen hat:

```
jshell> double x = 325e200
x ==> 3.25E202

jshell> double y = 325e-200
y ==> 3.25E-198

jshell> x
x ==> 3.25E202

jshell> y
y ==> 3.25E-198

jshell> x+y
$11 ==> 3.25E202

jshell> x+100000
$12 ==> 3.25E202
```

Wie man an der Ausgabe erkennen kann: selbst die Addition der Zahl `100000` bewirkt keine Veränderung auf einer großen Fließkommazahl:

Das folgende kleine Beispiel zeigt, inwieweit und für den Benutzer oft auf überraschende Weise die Fließkommadarstellung zu Rundungen führt:

```
jshell> 8f
$13 ==> 8.0

jshell> 88f
$14 ==> 88.0

jshell> 88888888f
$19 ==> 8.8888888E7

jshell> 888888888f
$20 ==> 8.888889E8

jshell> 8888888888888f
$24 ==> 8.8888889E12

jshell> 1f+1000000000000f-1000000000000f
$25 ==> 0.0
```

Insbesondere in de Auswertung des letzten Ausdrucks fällt auf, dass Addition und anschließende Subtraktion ein und derselben Zahl nicht die Identität ist. Für Fließkommazahlen gilt nicht: $x + y - y = x$.

Man vergleiche hierzu noch einmal die folgenden beiden Ausdrücke:

```
jshell> 1f+1000000000000f-1000000000000f
$25 ==> 0.0

jshell> 1f+(1000000000000f-1000000000000f)
$26 ==> 1.0
```

Merken Sie sich:

In Java gibt es exakt die folgenden acht primitiven Typen: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` und `double`. Alle andere Typen werden als Referenztypen bezeichnet.

3.1.3 Zeichen und Buchstaben

Wir haben bereits die Klasse String zur Darstellung von Texten in Form von Ketten einzelner Zeichen kennengelernt. Für einzelne Zeichen gibt es einen primitiven Datentypen. Einzelne Zeichen werden in einfachen Anführungszeichen notiert.

```
jshell> 'a'
$1 ==> 'a'

jshell> /var $1
|     char $1 = 'a'
```

Wie man sieht, heißt der entsprechende primitive Typ `char`.

Für manche besondere Zeichen gibt es sogenannte Escape-Sequenzen. Dieses sind z.B. das Zeilenende oder die Tabulatur. Diese Escape im Überblick:

Sequenz	**Bedeutung**
\t	Tabulatorzeichen
\b	Rückwärtsschritt
\n	Zeilenende
\r	Wagenrücklauf
\f	Seitenvorschub
\'	Einfaches Anführungszeichen
\"	Doppeltes Anführungszeichen
\\	Rückwärtiger Schrägstrich

Es lassen sich Zeichen nicht nur der für uns gängigen westeuropäischen Schriften darstellen sondern beliebiger gängiger Schriften weltweit. Hierzu wird Unicode verwendet, ein internationaler Standard, der für jedes Schriftzeichen aller Schriftkulturen ein Code festlegt. Wenn Sie im Java-Quelltext ein Zeichen nicht direkt eingeben können, weil Sie zum Beispiel keine Taste für ein chinesisches Schriftzeichen haben, dann können Sie in dem char-Literal die entsprechende Nummer der Unicode-Codierung angeben. Hierzu wird `\u` der hexadezimal codierten Unicode-Nummer vorangestellt.[3]

```
jshell> '\u0163'
$3 ==> 'ţ'

jshell> '\u0463'
$4 ==> 'ѣ'

jshell> '\uC463'
$5 ==> '[U+C463]'
```

Der primitive Typ `char` wird in 16-Bit im speicher abgelegt. In Unicode gibt es aber mittlerweile mehr verschiedene Zeichen, als in 16-Bit dargestellt werden können (z.B. die ägyptischen Hieroglyphen). Diese können dann nicht mehr in einem `char` gespeichert werden.

[3] Leider hat unser Textsatzprogramm nicht Fonts für alle Zeichen, so dass Sie in der gedruckten Version z.B. das chinesische Zeichen derzeit nicht angezeigt bekommen. Versuchen Sie es selbst mal in der jshell einzugeben.

Strings sind Folgen von `chars`. In einem Stringliteral können Sie wie in `char`-Literalen die Zeichen auch mit ihrer Unicode-Nummer angeben. Hier gibt es auch die Möglichkeit, Unicode-Zeichen, die mehr 16-Bit in der Codierung brauchen darzustellen.[4]

```
jshell> "\uC463\uC464\uC465"
$12 ==> "[U+C463][U+C464][U+C465]"

jjshell> "\uD835\uDD0A"
```

Man kann die Werte des Typs `char` auch als Zahlen speichern. Dann wird die Nummer aus dem Unicode-Standard hierfür verwendet.

```
jshell> int x = 'a'
x ==> 97
```

Damit lassen sich mit Werten des Typs `char` auch Vergleiche durchführen. Zum Beispiel kann man leicht testen, ob ein bestimmtes Zeichen in einem bestimmten Zeichenbereich liegt:

```
jshell> char klein(char c){
   ...>   if (c>='A' && c<='Z') return (char) (c-('A'-'a'));
   ...>   return c;
   ...> }
|  created method klein(char)
```

Obige Methode berechnet für alle Großbuchstaben der lateinischen Schrift den entsprechenden Kleinbuchstaben, ansonsten ist die Funktion die Identität.

```
jshell> klein('D')
$11 ==> 'd'

jshell> klein('e')
$12 ==> 'e'

jshell> klein('\uF342')
$13 ==> '[U+F342]'
```

In der Standardklasse `Character` finden sich eine Vielzahl nützlicher statischer Methoden, die auf `char`-Werten arbeiten.

[4] Das Ergebnis des zweiten der folgenden Ausdrücke ist leider mit unserem Textverarbeitungsprogramm nicht darstellbar.

```
jshell> Character.isDigit('9')
$26 ==> true

jshell> Character.isAlphabetic('9')
$27 ==> false
```

3.1.4 Boxen für primitive Typen

Javas Typsystem ist etwas zweigeteilt. Es gibt Objekttypen (auch Referenztypen genant) und primitive Typen. Die Daten der primitiven Typen stellen keine Objekte dar. Für jeden primitiven Typen gibt es allerdings eine Klasse, die es erlaubt, Daten eines primitiven Typs als Objekt zu speichern. Dieses sind die Klassen `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` und `Character`. Objekte dieser Klassen sind nicht modifizierbar. Einmal ein Integer-Objekt mit einer bestimmten Zahl erzeugt, lässt sich die in diesem Objekt erzeugte Zahl nicht mehr verändern.

Will man in einer Refernzvariablen daten eines primitiven Typen speichern, so muss man es in einem Objekt der entsprechenden Klasse kapseln. Man spricht von *boxing*. Es kommt zu Konstruktoraufrufen dieser Klassen. Sollen später mit Operatoren auf den Zahlen, die durch solche gekapselten Objekte ausgedrückt wurden, gerechnet werden, so ist der primitive Wert mit einem Methodenaufruf aus dem Objekt wieder zu extrahieren, dem sogenannten *unboxing*. Es kommt zu Aufrufen von Methoden wie `intValue()` im Code.

In diesem Beispiel sieht man das manuelle Verpacken und Auspacken primitiver Daten.

ManualBoxing.java

```
package name.panitz.boxing;
public class ManualBoxing{
  public static void main(String [] args){
    int i1 = 42;
    Integer i = new Integer(i1);
    System.out.println(i);
    Integer i2 = new Integer(17);
    Integer i3 = new Integer(4);
    int i4 = 21;
    System.out.println((i2.intValue()+i3.intValue())*i4);
  }
}
```

Dieses *boxing* und *unboxing* ist nicht manuell notwendig. In Java können die primitiven Typen mit ihren entsprechenden Klassen synonym verwendet werden. Nach außen hin werden die primitiven Typen auch zu Objekttypen. Der Übersetzer nimmt fügt die notwendigen *boxing*- und *unboxing*-Operationen vor.

Jetzt das vorherige kleine Programm ohne explizite *boxing*- und *unboxing*-Aufrufe.

AutomaticBoxing.java

```
package name.panitz.boxing;
public class AutomaticBoxing{
  public static void main(String [] args){
    int i1 = 42;
    Integer i = i1;
    System.out.println(i);
    Integer i2 = 17;
    Integer i3 = 4;
    int i4 = 21;
    System.out.println((i2+i3)*i4);
  }
}
```

3.2 Ausdrücke

Wir haben jetzt gesehen, was Java uns für Typen zur Darstellung von Zahlen zur Verfügung stellt. Jetzt wollen wir mit diesen Zahlen nach Möglichkeit auch noch rechnen können. Hierzu stellt Java eine feste Anzahl von Operatoren wie `*`, `-`, `/` etc. zur Verfügung. Prinzipell gibt es in der Informatik für Operatoren drei mögliche Schreibweisen:

- **Präfix:** Der Operator wird vor den Operanden geschrieben, also z.B. `(* 2 21)`. Im ursprünglichen Lisp gab es die Prefixnotation für Operatoren.
- **Postfix:** Der Operator folgt den Operanden, also z.B. `(21 2 *)`. Forth und Postscript sind Beispiele von Sprachen mit Postfixnotation.
- **Infix:** Der Operator steht zwischen den Operanden. Dieses ist die gängige Schreibweise in der Mathematik und für das Auge die gewohnteste. Aus diesem Grunde bedient sich Java der Infixnotation: `42 * 2`.

3.2.1 Arithmetische Operatoren

Java stellt für Zahlen die vier Grundrechenarten zur Verfügung.

Bei der Infixnotation gelten für die vier Grundrechenarten die üblichen Regeln der Bindung, nämlich Punktrechnung vor Strichrechnung. Möchte man diese Regel durchbrechen, so sind Unterausdrücke in Klammern zu setzen. Folgender kleiner Test demonstriert noch einmal den Unterschied:

```
jshell> 17 + 4 * 2
$28 ==> 25

jshell> (17 + 4) * 2
$29 ==> 42
```

Wir können nun also Methoden schreiben, die Rechnungen vornehmen. Sie erinnern sich an die Methode zum Berechnen der Quadratzahl des Arguments:

```
jshell> int square(int i){
   ...>   return i*i;
   ...> }
|  replaced method square(int)

jshell> square(17)
$32 ==> 289
```

Für die Division auf Ganzzahlen gibt es zwei Operatoren. Den eigentlichen Divisionsoperator `/` und den Operator `%`, der den ganzzahligen Rest einer Division anzeigt, der sogenannte Modulo-Operator:

```
jshell> 10/3
$33 ==> 3

jshell> 10%3
$34 ==> 1

jshell> -10/3
$35 ==> -3

jshell> -10%3
$36 ==> -1
```

3.2.2 Vergleichsoperatoren

Obige Operatoren rechnen jeweils auf zwei Zahlen und ergeben wieder eine Zahl als Ergebnis. Vergleichsoperatoren vergleichen zwei Zahlen und geben einen bool'schen Wert, der angibt, ob der Vergleich wahr oder falsch ist. Java stellt die folgenden Vergleichsoperatoren zur Verfügung: `<`, `<=`, `>`, `>=`, `!=`, `==`. Für die Gleichheit ist in Java das doppelte Gleichheitszeichen `==` zu schreiben, denn das einfache Gleichheitszeichen ist bereits für den Zuweisungsbefehl vergeben. Die Ungleichheit wird mit `!=` bezeichnet.

Folgende Tests in der jshell demonstrieren die Benutzung der Vergleichsoperatoren:

```
jshell> 1+1 < 42
$1 ==> true

jshell> 1+1 <= 42
$2 ==> true

jshell> 1+1 >  42
$3 ==> false

jshell> 1+1 >= 42
$4 ==> false

jshell> 1+1 == 42
$5 ==> false

jshell> 1+1 != 42
$6 ==> true
```

3.2.3 Logische Operatoren

In der bool'schen Logik gibt es eine ganze Reihe von binären Operatoren für logische Ausdrücke. Für zwei davon stellt Java auch Operatoren bereit: `&&` für das logische Und $\wedge$ und `||` für das logische Oder $\vee$.

Zusätzlich kennt Java noch den unären Operator der logischen Negation $\neg$. Er wird in Java mit `!` bezeichnet.

Wie man im folgenden Test sehen kann, gibt es auch unter den bool'schen Operatoren eine Bindungspräzedenz, ähnlich wie bei der Regel Punktrechnung vor Strichrechnung. Der Operator `&&` bindet stärker als der Operator `||`:

```
jshell> true && false
$29 ==> false

jshell> true || false
$30 ==> true

jshell> !true || false
$31 ==> false

jshell> true || true && false
$32 ==> true
```

In der formalen Aussagenlogik, die Ihnen im Modul »Diskrete Strukturen« näher gebracht wird, kennt man noch weitere Operatoren, z.B. die Implikation $\rightarrow$. Diese Operatoren lassen sich aber durch die in Java zur Verfügung stehenden Operatoren ausdrücken.

$A \rightarrow B$ entspricht $\neg A \vee B$. Wir können somit eine Methode schreiben, die die logische Implikation testet:

```
jshell> boolean implication(boolean a, boolean b){
   ...>     return !a || b;
   ...>   }
|  created method implication(boolean,boolean)

jshell> implication(true,false)
$34 ==> false

jshell> implication(true,true)
$35 ==> true

jshell> implication(false,true)
$36 ==> true
```

3.2.4 Der Bedingungsoperator

Java kennt auch einen Operator mit drei Operanden. Er besteht aus zwei einzelnen Zeichen, die als Trenner zwischen den Operanden stehen. Zunächst kommt der erste Operand, dann das Zeichen `?`, dann der zweite Operand, gefolgt vom Zeichen `:`, dem der dritte Operand folgt. Schematisch sehen die Ausdrücke dieses Operators wie folgt aus:

$cond\ ?\ alt_1\ :\ alt_2$

Das Ergebnis dieses Operators wird wie folgt berechnet: Der erste Operand wird zu einem Wahrheitswert ausgewertet. Wenn dieser *wahr* ist, so wird der zweite Operand als Ergebnis ausgewertet, wenn er *falsch* ist, wird der dritte Operand als Ergebnis ausgewertet.

```
jshell> (17+4)*2==42?"tatsaechlich gleich":"unterschiedlich"
$1 ==> "tatsaechlich gleich"
```

Der Bedingungsoperator ist unser erstes Konstrukt, um Verzweigungen auszudrücken. Da der Bedingungsoperator auch einen Wert errechnet, können wir diesen benutzen, um mit ihm weiterzurechenen. Der Bedingungsoperator kann also tatsächlich ineinandergesteckt werden:

```
jshell> "signum(42) = "+((42>0)?1:((42<0)?-1:0))
$2 ==> "signum(42) = 1"
```

Hier wird zunächst geschaut, ob die Zahl 42 größer als 0 ist. Ist dieses der Fall wird die Zahl 1 ausgegeben, ansonsten wird weitergeschaut, ob die Zahl 42 kleiner 1 ist. Hier wird im Erfolgsfall die Zahl -1 ausgegeben. Wenn beides nicht der Fall war, wird die Zahl 0 ausgegeben.

Zugegebener Maßen ist dieser Ausdruck schon schwer zu lesen. Wir werden später bessere Konstrukte kennenlernen, um verschiedene Fälle zu unterscheiden.

Merken Sie sich:

Ausdrücke können immer zu einen Wert ausgewertet werden, sie haben immer ein Ergebnis.

3.3 Anweisungen

Im letzten Abschnitt haben wir Ausdrücke in Java bezeichnet. Ausdrücke sind Rechnungen, die immer zu einen Ergebnis ausgewertet werden können. Ein Ausdruck kann überall dort verwendet werden, wo Daten zum Weiterverarbeiten benötigt werden.

Zusätzlich gibt es in Java Anweisungen. Dieses sind meist komplexere strukturierte Befehle, in denen es an mehreren Stellen Ausdrücke gibt. Anweisungen haben kein Ergebnis, zu dem sie ausgewertet werden, sondern verändern Speicherzellen, indem Variablen neue Werte zugewiesen werden, oder Felder in Objekten verändert werden und steuern den Kontrollfluss eines Programms. Damit ist gemeint, dass sie z.B. Verzweigungen und Wiederholungen im Programmdurchlauf kontollieren.

3.3.1 Zuweisungen

Wir haben schon mehrfach gesehen, dass es in Java Variablen gibt. Wenn man in eine Variable Daten speichert, so nennt man dieses eine Zuweisung. Jede Variable hat einen unveränderbaren festen Typ. Daten die einer Variablen zugewiesen werden, müssen von diesem Typ sein, ansonsten gibt es eine Fehlermeldung vom Kompilator. Zur Zuweisung wird das Symbol des einfachen Gleichheitszeichens = verwendet. Auf der linken Seite einer Zuweisungsanweisung steht eine Variable, auf der rechten Seite ein Ausdruck, dessen Ergebnis in der Variablen gespeichert werden soll.

Folgendes Beispiel zeigt, dass eine nicht typgerechte Zuweisung zu einem Übersetzungsfehler führt:

```
jshell> String s = "hallo"
s ==> "hallo"

jshell> s = 42
|  Error:
|  incompatible types: int cannot be converted to java.lang.String
|  s = 42
|      ^^
```

Variablen, die mit Schlüsselwort `final` gekennzeichnet sind, dürfen nur einmal einen Wert zugewiesen bekommen. Sie sind somit Konstanten.[5]

Man kann auf der rechten Seite einer Zuweisung die Variablen, der ein neuer Wert zugewiesen werden soll, verwenden. Dann erhält man dort den Wert vor der neuen Zuweisung.

```
jshell> var x = 2
x ==> 2

jshell> x = x*2
x ==> 4
```

Hierzu gibt es eine Kurzschreibweise. `*=` steht zum Beispiel für: verdoppel den Wert, der in der Variablen `x` gespeichert ist.

```
jshell> x *= 2
$57 ==> 8

jshell> x *= x
$58 ==> 64
```

Es gibt noch zwei ganz spezielle Arten der Neuzuweisung eines Wertes einer Variablen. Dieses sind die Operatoren `++` und `--`. Sie erhöhen eine Variable um eins bzw. verringern eine Variable um 1. Diese Operatoren können der zu verändernen Variablen voran oder hintan gestellt werden

```
jshell> x--
$59 ==> 64

jshell> var z = 10
z ==> 10

jshell> --z
$68 ==> 9
```

Die Zuweisungen sind allesamt nicht nur Anweisungen, die eine Variable verändern, sondern können auch als Ausdrücke verwendet werden. Eine Zuweisung hat nämlich einen Wert. das Ergebnis der Zuweisung ist der neue gesetzte Wert der Variablen.

Hier gibt es eine Ausnahme. Es ist ein Unterschied, ob die Operatoren `++` und `--` präfix oder postfix verwendet werden.

Bei einer Postfixverwendung ist der Wert des Ausdrucks der Wert der Variablen vor der Veränderung.

[5] Variablen der jshell können nicht als `final` deklariert werden

```
jshell> x=2
x ==> 2

jshell> square(x++)
$61 ==> 4

jshell> x
x ==> 3
```

Bei einer Präfixverwendung ist der Wert des Ausdrucks der Wert der Variablen nach der Veränderung, also der neue Wert der Variablen.

```
jshell> square(++x)
$63 ==> 16

jshell> var y = 42
y ==> 42
```

Die Eigenschaft, dass eine Zuweisung auch ein Ausdruck mit einem Wert darstellt, kann verwirrend sein. Man sollte es besser nie oder selten verwenden.

```
jshell> x = y=17
x ==> 17

jshell> y
y ==> 17
```

3.3.2 Fallunterscheidungen

Bedingungsabfrage mit `if`

Ein häufig benötigtes Konstrukt ist, dass ein Programm abhängig von einer bool'schen Bedingung sich verschieden verhält. Hierzu stellt Java die **if**-Bedingung zur Verfügung. Dem Schlüsselwort `if` folgt in Klammern eine bool'sche Bedingung, anschließend kommen in geschweiften Klammern die Befehle, die auszuführen sind, wenn die Bedingung wahr ist. Anschließend kann optional das Schlüsselwort `else` folgen mit den Befehlen, die andernfalls auszuführen sind:

```
jshell> void firstIf(boolean bedingung){
   ...>     if (bedingung) {
   ...>       System.out.println("Bedingung ist wahr");
   ...>     } else {
   ...>       System.out.println("Bedingung ist falsch");
   ...>     }
   ...>   }
|  created method firstIf(boolean)

jshell> firstIf(17+4*2==42)
Bedingung ist falsch

jshell> firstIf(17+4*2==25)
Bedingung ist wahr
```

Das `if`-Konstrukt erlaubt es uns also, Fallunterscheidungen zu treffen. Wenn in den Alternativen nur ein Befehl steht, so können die geschweiften Klammern auch fortgelassen werden. Unser Beispiel läßt sich also auch schreiben als:

```
jshell> void firstIf(boolean bedingung){
   ...>     if (bedingung) System.out.println("Bedingung ist wahr");
   ...>     else System.out.println("Bedingung ist falsch");
   ...>   }
|  modified method firstIf(boolean)

jshell> firstIf(true||false)
Bedingung ist wahr
```

Eine Folge von mehreren `if`-Konstrukten läßt sich auch direkt hintereinanderschreiben, so dass eine Kette von `if`- und `else`-Klauseln entsteht:

```
jshell>  String lessOrEq(int i,int j){
   ...>    if (i<10)  return "i kleiner zehn";
   ...>    else if (i>10)  return "i größer zehn";
   ...>    else if (j<10)  return "j kleiner zehn";
   ...>    else if (j>10)  return "j größer zehn";
   ...>    else return "x == y == 10";
   ...> }
|  modified method lessOrEq(int,int)

jshell> lessOrEq(9,10)
$46 ==> "i kleiner zehn"

jshell> lessOrEq(10,11)
$47 ==> "j größer zehn"
```

Wenn zuviele `if`-Bedingungen in einem Programm einander folgen und ineinander ver-

schachtelt sind, dann wird das Programm schnell unübersichtlich. Man spricht auch von *Spaghetti-code*. In der Regel empfiehlt es sich, in solchen Fällen noch einmal über das Design nachzudenken, ob die abgefragten Bedingungen sich nicht durch verschiedene Klassen mit eigenen Methoden darstellen lassen.

Fallunterscheidungen mit `switch`

Aus C erbt Java eine sehr spezielle zusammengesetzte Anweisung, die **switch**-Anweisung. Es ist eine Anweisung für eine Fallunterscheidung mit mehreren Fällen, die `switch`-Anweisung. Die Idee dieser Anweisung ist, eine Kette von mehreren `if-then`-Anweisungen zu vermeiden. Leider ist die `switch`-Anweisung in seiner Anwendungsbereite recht begrenzt und in Form und Semantik ziemlich veraltet.

Schematisch hat die `switch`-Anweisung die folgende Form:

```
switch (expr){
  case const: stats
    ...
  case const: stats
  default: stats
}
```

Dem Schlüsselwort `switch` folgt ein Ausdruck, nach dessen Wert eine Fallunterscheidung getroffen werden soll. In geschweiften Klammern folgen die verschiedenen Fälle. Ein Fall beginnt mit dem Schüsselwort `case` gefolgt von einer Konstante. Diese Konstante ist von einem ganzzahligen Typ und darf kein Ausdruck sein, der erst während der Laufzeit berechnet wird. Es muss hier eine Zahl stehen. Die Konstante muss während der Übersetzungszeit des Programms feststehen. Der Konstante folgt ein Doppelpunkt, dem dann die Anweisungen für diesen Fall folgen. Ein besonderer Fall ist der `default`-Fall. Dieses ist der Standardfall. Er wird immer ausgeführt, egal was für einen Wert der Ausdruck nach dem die `switch`-Anweisung unterscheidet hat.

Ein kleines Beispiel soll die operationale Semantik dieser Anweisung verdeutlichen.

```
jshell> void sw1(int i){
   ...>   switch (4*i){
   ...>       case 42 : System.out.println(42);
   ...>       case 52 : System.out.println(52);
   ...>       case 32 : System.out.println(32);
   ...>       case 22 : System.out.println(22);
   ...>       case 12 : System.out.println(12);
   ...>       default : System.out.println("default");
   ...>     }
   ...> }
|  created method sw1(int)
```

Der Aufruf mit dem Wert `13`, so dass der Ausdruck, nach dem wir die Fallunterscheidung durchführen, zu `52` auswertet, so führt alle Ausdrucke auf der Kommandozeile aus:

```
jshell> sw1(13)
52
32
22
12
default
```

Wie man sieht, springt die `switch`-Anweisung zum Fall für den Wert `52`, führt aber nicht nur dessen Anweisungen aus, sondern alle folgenden Anweisungen.

der Aufruf mit dem Wert 3 führt entsprechend zu folgendem Verhalten:

```
jshell> sw1(3)
12
default
```

Das oben beobachtete Verhalten ist verwirrend. Zumeist will man in einer Fallunterscheidung, dass nur die entsprechenden Anweisung für den vorliegenden Fall ausgeführt werden und nicht auch für alle folgenden Fälle. Um dieses zu erreichen, gibt es die `break`-Anweisung, wie wir sie auch schon von den Schleifenanweisungen kennen. Endet man jeden Fall mit der `break`-Anweisung, dann erhälz man das meistens erwünschte Verhalten.

Das obige Beispiel läßt sich durch Hinzufügen der `break`-Anweisung so ändern, dass immer nur ein Fall ausgeführt wird.

```
jshell> void sw2(int i){
   ...>   switch (4*i){
   ...>       case 42 : System.out.println(42);break;
   ...>       case 52 : System.out.println(52);break;
   ...>       case 32 : System.out.println(32);break;
   ...>       case 22 : System.out.println(22);break;
   ...>       case 12 : System.out.println(12);break;
   ...>       default : System.out.println("default");
   ...>     }
   ...>   }
|  modified method sw2(int)

jshell> sw2(13)
52

jshell> sw2(3)
12
```

An der Ausgabe sehen wir, dass zu einem Fall gesprungen wird und am Ende dieses Falls die Anweisung verlassen wird.

`switch`-Ausdrücke ab Java 12 und 13 Die `switch`-Anweisung erfährt ab Version 12 von Java eine wesentliche Verbesserung, die zunächst einmal auf die etwas verwirrende operationale Semantik abzielt. Ab Java 12, wenn die neue Konstrukte anschaltet, gibt es eine neue Variante, der `switch`-Anweisung. Statt eines Doppelpunktes steht nach den einzelnen `case`-Klauseln ein Pfeil in Form von `->`. Die obige Funktion `sw1` sieht dann wie folgt aus:

```
jshell  --enable-preview
|  Welcome to JShell -- Version 13
|  For an introduction type: /help intro

jshell> void sw1(int i){
   ...>   switch (4*i){
   ...>     case 42 -> System.out.println(42);
   ...>     case 52 -> System.out.println(52);
   ...>     case 32 -> System.out.println(32);
   ...>     case 22 -> System.out.println(22);
   ...>     case 12 -> System.out.println(12);
   ...>     default -> System.out.println("default");
   ...>   }
   ...> }
|  created method sw1(int)
```

Jetzt verhält sich die Anweisung tatsächlich so, dass zu einem Fall gesprungen wird und genau dieser ausgeführt wird und nicht noch die nachfolgenden Fälle. Es ist keine `break`-Anweisung mehr notwendig.

```
jshell> sw1(13)
52

jshell> sw1(3)
12
```

Enden die `case`-Klauseln alle mit einem Ausdruck (gleichen Typs), dann wird kann das neue `switch`-Konstrukt sogar als Ausdruck verwendet werden. Dann hat die Ausführung eines `switch`-Ausdrucks immer einen Wert, der dann in anderen Anweisungen verwendet werden kann. In folgender Funktion z.B. wird der der `switch`-Ausdruck als Ausdruck einer `return`-Anweisung verwendet.

```
jshell> String digitString(int i){
   ...>   return
   ...>     switch (i){
   ...>       case 0 -> "null";
   ...>       case 1 -> "eins";
   ...>       case 2 -> "zwei";
   ...>       case 3 -> "drei";
   ...>       case 4 -> "vier";
   ...>       case 5 -> "fünf";
   ...>       case 6 -> "sechs";
   ...>       case 7 -> "sieben";
   ...>       case 8 -> "acht";
   ...>       case 9 -> "neun";
   ...>       default -> "keine Ziffer";
   ...>   };
   ...> }
|  created method digitString(int)
```

Die Funktion liefert entsprechend einen String als Ergebnis:

```
jshell> digitString(2+3)
$5 ==> "fünf"

jshell> digitString(2+13)
$6 ==> "keine Ziffer"
```

Es ist angedacht dieses in der Zukunft auf ein volles Pattern-Matching auf reinen Datenhaltungsklassen zu erweitern. Als Datenhaltungsklassen sind Klassen zu verstehen, die keine Methoden haben, sondern nur Felder und einen Konstruktor, der diese Felder initialisiert.

Diese komplette Erweiterung von pattern-matching in Java wird wahrscheinlich frühestens mit der nächsten Hauptversion von Java kommt (wahrscheinlich Java17).

3.3.3 Iteration

Sehr häufig ist zu programmieren, dass ein bestimmter Code-Block mehrfach wiederholt durchlaufen werden soll. Der Fachausdruck für eine solche Wiederholung heißt *Iteration*. Viele Programmiersprache[6] bieten hierfür spezielle Anweisungen in Form von Schleifen an.

[6] Fast alle aber nicht alle. Ein Gegenbeispiel ist Haskell.

Schleifen mit `while`

Die einfache Schleifenanweisung in Java ist die while-Schleife, die abhängig von einer Bedingung einen Code-Block mehrfach wiederholt durchläuft. Sie kommt in zwei Varianten daher.

Vorgeprüfte Schleifen Die vorgeprüften Schleifen haben folgendes Schema in Java:

`while (`*pred*`){`*body*`}`

pred ist hierbei ein Ausdruck, der zu einem bool'schen Wert auswertet. *body* ist eine Folge von Befehlen. Java arbeitet die vorgeprüfte Schleife ab, indem erst die Bedingung *pred* ausgewertet wird. Ist das Ergebnis `true`, dann wird der Rumpf *(body)* der Schleife durchlaufen. Anschließend wird wieder die Bedingung geprüft. Dieses wiederholt sich so lange, bis die Bedingung zu `false` auswertet.

Ein einfaches Beispiel ist eine Schleife, deren Bedingung nie zu `false` ausgewertet wird. Eine solche Schleife wird unendlich oft durchlaufen.

```
jshell> while (true){
   ...>   System.out.println("young");
   ...> }
young
young
young
young
young
...
```

Wir können eine Variable einführen, die bei jedem Schleifendurchlauf um eins erhöht wird. Somit zählt die Variable, wie oft die Schleife durchlaufen wird.

```
jshell>  int i = 0;
i ==> 0

jshell> while (true){
   ...>    System.out.println("young: "+i);
   ...>    i = i + 1;
   ...>  }
young: 0
young: 1
young: 2
young: 3
young: 4
...
```

Diese Variable kann nun dazu genutzt werden, die Schleifen nach einer bestimmten Anzahl von Durchläufen zu beenden, in dem die Schleifenbedingung dieses ausdrückt.

```
jshell>  int i = 0;
i ==> 0

jshell> while (i < 5){
   ...>   System.out.println("young: "+i);
   ...>   i = i + 1;
   ...> }
young: 0
young: 1
young: 2
young: 3
young: 4
```

Wie man an den Beispielen sieht, gibt es oft eine lokale Variable, die zur Steuerung der Schleife benutzt wird. Dieses verändert innerhalb des Schleifenrumpfes seinen Wert. Abhängig von diesem Wert wird die Schleifenbedingung beim nächsten Bedingungstest wieder wahr oder falsch. Eine Variable, von der die Schleifenbedingung abhängt und die im Schleifenrumpf verändert wird, bezeichnet man als Schleifenvariable.

Nun kann man mit einer Schleife nach und nach ein Ergebnis in Abhängigkeit von der Schleifenvariable errechnen. Folgende Methode errechnet die Summe aller Zahlen in einem bestimmten Zahlenbereich.

```
jshell> int summe(int n){
   ...>  int erg = 0 ;           // Ergebnisvariable.
   ...>  int j   = n;            // Schleifenvariable.
   ...>
   ...>   while (j>0){           // j läuft von n bis 1.
   ...>     erg = erg + j;       // akkumuliere das Ergebnis.
   ...>     j = j-1;             // verringere Laufzähler.
   ...>   }
   ...>
   ...>   return erg;
   ...> }
|  created method summe(int)

jshell> summe(4)
$21 ==> 10

jshell> summe(20)
$22 ==> 210

jshell> summe(100)
$23 ==> 5050
```

Nachgeprüfte Schleifen In der zweiten Variante der `while`-Schleife steht die Schleifenbedingung syntaktisch nach dem Schleifenrumpf:

`do {`*body*`} while (`*pred*`)`

Bei der Abarbeitung einer solchen Schleife wird entsprechend der Notation, die Bedingung erst nach der Ausführung des Schleifenrumpfes geprüft. Am Ende wird also geprüft, ob die Schleife ein weiteres Mal zu durchlaufen ist. Das impliziert insbesondere, dass der Rumpf mindestens einmal durchlaufen wird.

```
jshell> do {
   ...>   System.out.println(i);
   ...>   i = i+1;
   ...> } while (i < 5);
0
1
2
3
4
```

Man kann sich leicht davon vergewissern, dass die nachgeprüfte Schleife mindestens einmal durchlaufen[7] wird:

```
jshell> boolean falsch(){return false;}
|  created method falsch()

jshell> while (falsch()) System.out.println("vorgeprüfte Schleife")

jshell>
```

In diesen Fall wurde die Schleife kein Mal durchlaufen.

```
jshell> do {System.out.println("nachgeprüfte Schleife");}
   ...>   while (falsch());
nachgeprüfte Schleife

jshell>
```

In diesem Fall zumindest ein Mal.

Schleifen mit `for`

Das syntaktisch aufwendigste Schleifenkonstrukt in Java ist die *for*-Schleife.

[7] Der Javaübersetzer macht kleine Prüfungen auf konstanten Werten, ob Schleifen jeweils durchlaufen werden oder nicht terminieren. Deshalb brauchen wir die Hilfsmethode `falsch()`.

Wer sich die obigen Schleifen anschaut, sieht, dass sie an drei verschiedenen Stellen im Programmtext Code haben, der kontrolliert, wie oft die Schleife zu durchlaufen ist. Oft legen wir ein spezielles Feld an, dessen Wert die Schleife kontrollieren soll. Dann gibt es im Schleifenrumpf einen Zuweisungsbefehl, der den Wert dieses Feldes verändert. Schließlich wird der Wert dieses Feldes in der Schleifenbedingung abgefragt.

Die Idee der *for*-Schleife ist, diesen Code, der kontrolliert, wie oft die Schleife durchlaufen werden soll, im Kopf der Schleife zu bündeln. Solche Daten sind oft Zähler vom Typ `int`, die bis zu einem bestimmten Wert herunter oder hoch gezählt werden. Später werden wir noch die Standardklasse `Iterator` kennenlernen, die benutzt wird, um durch Listenelemente durchzuiterieren.

Eine *for*-Schleife hat im Kopf

- eine Initialisierung der relevanten Schleifensteuerungsvariablen *(init)*,
- ein Prädikat als Schleifenbedingung *(pred)*
- und einen Befehl, der die Schleifensteuerungsvariable weiterschaltet *(step)*.

`for (`*init*`;`*pred*`;`*step*`){`*body*`}`

Entsprechend sieht unsere jeweilige erste Schleife (die Ausgabe der Zahlen von 0 bis 9) in der `for`-Schleifenversion wie folgt aus:

```
jshell> for (int i=0; i<5; i=i+1){
   ...>   System.out.println(i);
   ...> }
0
1
2
3
4
```

Die Reihenfolge, in der die verschiedenen Teile der `for`-Schleife durchlaufen werden, wirkt erst etwas verwirrend, ergibt sich aber natürlich aus der Herleitung der `for`-Schleife aus der vorgeprüften `while`-Schleife:

Als erstes wird genau einmal die Initialisierung der Schleifenvariablen ausgeführt. Anschließend wird die Bedingung geprüft. Abhängig davon wird der Schleifenrumpf ausgeführt. Als letztes wird die Weiterschaltung ausgeführt, bevor wieder die Bedingung geprüft wird.

Die nun schon hinlänglich bekannte Methode `summe` stellt sich in der Version mit der `for`-Schleife wie folgt dar:

```
jshell> int summe(int n){
   ...>   int erg = 0 ;                    // Feld für Ergebnis
   ...>   for (int j = n;j>0;j=j-1){       // j läuft von n bis 1
   ...>     erg = erg + j;                 // akkumuliere das Ergebnis
   ...>   }
   ...>   return erg;
   ...> }
|  modified method summe(int)

jshell> summe(4)
$32 ==> 10

jshell> summe(20)
$33 ==> 210

jshell> summe(100)
$34 ==> 5050
```

Beim Vergleich mit der `while`-Version erkennt man, wie sich die Schleifensteuerung im Kopf der `for`-Schleife nun gebündelt an einer syntaktischen Stelle befindet.

Die drei Teile des Kopfes einer `for`-Schleife können auch leer sein. Dann wird in der Regel an einer anderen Stelle der Schleife entsprechender Code zu finden sein. So können wir die Summe auch mit Hilfe der `for`-Schleife so schreiben, dass die Schleifeninitialisierung und Weiterschaltung vor der Schleife bzw. im Rumpf durchgeführt wird:

```
jshell> int summe(int n){
   ...>   int erg = 0 ;              // Feld für Ergebnis.
   ...>   int j   = n;               // Feld zur Schleifenkontrolle
   ...>
   ...>   for (;j>0;){               // j läuft von n bis 1
   ...>     erg = erg + j;          // akkumuliere das Ergebnis.
   ...>     j = j-1;                // verringere Laufzähler
   ...>   }
   ...>   return erg;
   ...> }
|  modified method summe(int)

jshell> summe(100)
$36 ==> 5050
```

Wie man jetzt sieht, ist die `while`-Schleife nur ein besonderer Fall der `for`-Schleife. Obiges Programm ist ein schlechter Programmierstil. Hier wird ohne Not die Schleifensteuerung mit der eigentlichen Anwendungslogik vermischt.

Schleifen innerhalb von Schleifen

Oft kommt man mit einer Schleife allein nicht aus. Wann immer man einen 2-dimensionalen Raum durchlaufen will, gibt es zwei Richtungen, die x-Richtung und die y-Richtung. Es ist dann durch alle (x,y)-Paare zu iterieren. Hierzu sind zwei ineinander verschachtelte Schleifen notwendig. Die äußere Schleife durchläuft dabei die ein Dimension, die innere Schleife die zweite.

Folgendes Beispiel gibt ein kleines Viereck auf der Kommandozeile aus. Hierfür gibt es eine äußere Schleife, die dafür sorgt, dass jede Zeile ausgegeben wird und eine innere Schleife, die für jede Zeile die einzelnen Zeichen ausgibt.

```
jshell> String mkCross(int width){
   ...>   String result = "";
   ...>   for (int y = 0; y < width; y = y +1){
   ...>     for (int x = 0; x < width; x = x +1){
   ...>       result = result + ( (x==y || (x+y) == (width-1)) ? 'X' : 'O' );
   ...>     }
   ...>     result = result + '\n';
   ...>   }
   ...>   return result;
   ...> }
|  created method mkCross(int)

jshell> System.out.println(mkCross(5))
XOOOX
OXOXO
OOXOO
OXOXO
XOOOX
```

Vorzeitiges Beenden von Schleifen

Java bietet innerhalb des Rumpfes seiner Schleifen zwei Befehle an, die die eigentliche Steuerung der Schleife durchbrechen. Entgegen der im letzten Abschnitt vorgestellten Abarbeitung der Schleifenkonstrukte, führen diese Befehle zum plötzlichen Abbruch des aktuellen Schleifendurchlaufs.

Verlassen der Schleife Der Befehl, um eine Schleife komplett zu verlassen, heißt `break`. Der `break` führt zum sofortigen Abbruch der nächsten äußeren Schleife.

Der `break`-Befehl wird in der Regel mit einer `if`-Bedingung auftreten.

Mit diesem Befehl lässt sich die Schleifenbedingung auch im Rumpf der Schleife ausdrücken. Das Programm der Zahlen 0 bis 9 lässt sich entsprechend unschön auch mit Hilfe des `break`-Befehls wie folgt schreiben.

```
jshell> int i = 0
i ==> 0

jshell> while (true){
   ...> if (i>9) {break;};
   ...>   i = i+1;
   ...>   System.out.println(i);
   ...> }
1
2
3
4
5
6
7
8
9
10
```

Gleichfalls läßt sich der `break`-Befehl in der `for`-Schleife anwenden. Dann wird der Kopf der `for`-Schleife vollkommen leer:

```
jshell> int i = 0
i ==> 0

jshell> for (;;){
   ...>   if (i>9) break;
   ...>   System.out.println(i);
   ...>   i=i+1;
   ...> }
0
1
2
3
4
5
6
7
8
9
```

In der Praxis wird der `break`-Befehl gerne für besondere Situationen inmitten einer längeren Schleife benutzt, z.B. für externe Signale.

Verlassen des Schleifenrumpfes Die zweite Möglichkeit, den Schleifendurchlauf zu unterbrechen, ist der Befehl `continue`. Diese Anweisung bricht nicht die Schleife komplett ab, sondern nur den aktuellen Durchlauf. Es wird zum nächsten Durchlauf gesprungen.

Folgendes kleines Programm druckt mit Hilfe des `continue`-Befehls die Zahlen aus, die durch 17 oder 19 teilbar sind:

```
jshell> for (int i=1; i<1000;i=i+1){
   ...>   // wenn nicht die Zahl durch 17 oder 19 ohne Rest teilbar ist
   ...>   if (!(i % 17 == 0 || i % 19 == 0)) continue;
   ...>   System.out.println(i);
   ...> }
17
19
34
38
...
```

Wie man an der Ausgabe dieses Programms sieht, wird mit dem Befehl `continue` der Schleifenrumpf verlassen und die Schleife im Kopf weiter abgearbeitet. Für die `for`-Schleife heißt das insbesondere, daß die Schleifenweiterschaltung der nächste Ausführungsschritt ist.

Verlassen einer äußeren Schleife durch Labels Die Anweisungen `break` und `continue` zum außerplanmäßigen Verlassen einer Schleife bzw. eines Schleifendurchlaufs, beziehen sich immer auf die zuletzt begonnene Schleife. In seltenen Situationen möchte man aber bei verschachtelten Schleifen nicht die innere sondern eine der äußeren Schleifen verlassen. Hierzu kann man Labels verwenden. Ein Label ist ein beliebiger Bezeichner der mit einem Doppelpunkt getrennt vor einer Schleife steht. Damit wird einer Schleife quasi ein Name gegeben. bei den Anweisungen `break` und `continue` kann nun dieser Name der Schleife mit angegeben werden. Somit lassen sich auch äußere Schleifen unterbrechen.

```
jshell> aussen:for (int x=1;x<=5;x++){
   ...>   for (int y=1;y<=5;y++){
   ...>     if (x==y)continue aussen;
   ...>     System.out.println("("+x+","+y+")");
   ...>   }
   ...> }
(2,1)
(3,1)
(3,2)
(4,1)
(4,2)
(4,3)
(5,1)
(5,2)
(5,3)
(5,4)
```

Tatsächlich habe ich persönlich in über 20 Jahren Javaprogrammierung von dieser Mög-

lichkeit noch nie Gebrauch gemacht und kenne sie in Projekten nur aus automatisch generierten Code.

Nachvollziehen von Anweisungen Komplexere Anweisungen, die mehrfach verschachtelt sind und in denen verschiedene Variablen an unterschiedlichen Stellen durch Zuweisungen verändert werden, sind schwer nachzuvollziehen. Hier empfiehlt es sich den dynamischen Prozess der Programmauswertung schrittweise von Hand durchzuführen und dabei darüber in einer Tabelle Buch zu führen, wie sich die Werte der Variablen verändern.

Betrachten Sie dazu folgendes kleines Programmfragment:

```
jshell> int x = 25;
x ==> 25

jshell> for (int y = 6;y<=x-2;x--){
   ...>   switch (x-1){
   ...>     case 20:;
   ...>     case 19:;
   ...>     case 42:;
   ...>     case 9: x=x-y-2;
   ...>     case 21: y--;
   ...>       break;
   ...>     default: x--;
   ...>   }
   ...>   x -=1;
   ...>   System.out.println(x+" "+y);
   ...> }

jshell>
```

Es gibt darin zwei Variablen und innerhalb einer for-Schleife noch eine switch-case-Anweisung. zusätzlich werden Ausgaben auf der Kommandozeile gemacht.

x	y	Ausgabe
25		
25	6	
24	6	
23	6	`"23 6"`
22	6	
22	5	
21	5	`"21 5"`
20	5	
13	5	
13	4	
12	4	`"12 4`
12	4	
11	4	
10	4	
9	4	
9	4	`"9 4"`
8	4	
7	4	
6	4	`"6 4"`

Wem das manuelle Nachvollziehen von Code zu mühsam ist, der hat zweierlei Möglichkeiten:

- Einfügen von Ausgaben oder Logging-Anweisungen im Code.
- Verwendung eines Debugger-Werkzeuges, mit dem schrittweise der Quelltext ausgeführt werde kann.

Beides ist eine Möglichkeit, um das Verhalten eines Programms nachzuvollziehen und eventuell dabei inhaltliche Fehler zu entdecken.

Merken Sie sich:

Im Gegensatz zu Ausdrücken habe Anweisungen kein Ergebnis sondern steuern den Kontrollfluss eines Programms. Innerhalb von Anweisungen werden Ausdrücke verwendet.

3.4 Rekursion

Sobald wir die Signatur einer Funktion oder Prozedur definiert haben, dürfen wir sie benutzen, sprich aufrufen. Damit ergibt sich eine sehr mächtige Möglichkeit der Programmierung. Wir können Funktionen bereits in ihren eigenen Rumpf aufrufen. Solche Funktionen werden rekursivRekursion genannt. *Recurrere* ist das lateinische Wort für

zurücklaufen. Eine rekursive Funktion läuft während ihrer Auswertung wieder zu sich selbst zurück.

Damit lassen sich wiederholt Programmteile ausführen. Das folgende Programm wird z.B. nicht müde, uns mit dem Wort `young` zu erfreuen.

```
jshell> void foreverYoung(){
   ...>   System.out.println("young");
   ...>   foreverYoung();
   ...> }
|  created method foreverYoung()

jshell> foreverYoung()
young
young

...

young
|  Exception java.lang.StackOverflowError
|        at UTF_8$Encoder.encodeLoop (UTF_8.java:564)
|        at CharsetEncoder.encode (CharsetEncoder.java:576)
|        at StreamEncoder.implWrite (StreamEncoder.java:292)
|
```

Der Aufruf der Funktion `foreverYoung` druckt einmal das Wort `young` auf die Konsole und ruft sich dann selbst wieder auf. Dadurch wird wieder `young` auf die Konsole geschrieben und so weiter. Wir haben ein endlos laufendes Programm. Tatsächlich endlos? Nein, es bricht nach kurzer Zeit mit einer Fehlermeldung ab.

Was zunächst wie eine Spielerei anmutet, kann verfeinert werden, indem mit Hilfe eines Arguments mitgezählt wird, wie oft die Prozedur bereits rekursiv aufgerufen wurde:

```
jshell> void halloZaehler(int i){
   ...>   System.out.println("hallo "+i);
   ...>   halloZaehler(i+1);
   ...> }
|  created method halloZaehler(int)

jshell>
```

Auch diese Methode läuft endlos.

Wie auch bereits bei Schleifen können wir über eine Bedingung dafür sorgen, dass der rekursive Aufruf nur unter bestimmten Umständen ausgeführt werden soll und damit dafür sorgen, dass die Rekursion irgend einmal terminiert.

```
jshell> void hallo(){
   ...>   System.out.println("hallo");
   ...> }
|  modified method hallo()

jshell> void nMalHallo(int n) {
   ...>   if (n>0){
   ...>     hallo();
   ...>     nMalHallo(n-1);
   ...>   }
   ...> }
|  modified method nMalHallo(int)

jshell> nMalHallo(4)
hallo
hallo
hallo
hallo

jshell>
```

Nach soviel esoterisch anmutenden Ausflügen in die Theorie, wollen wir nun auch eine rekursive Funktion schreiben, die etwas interessantes berechnet. Hierzu schreiben wir einmal die Fakultätsfunktion, die mathematisch definiert ist als:

$$fac(n) = \begin{cases} 1 & \text{für } n \leq 0 \\ n * fac(n-1) & \text{für } n > 0 \end{cases}$$

Diese Definition lässt sich direkt in ein Java Programm umsetzen:

```
jshell> long fac(long n){
   ...>   if (n<=0) return 1;
   ...>   return n*fac(n-1);
   ...> }
|  created method fac(long)

jshell> fac(5)
$51 ==> 120

jshell> fac(10)
$52 ==> 3628800

jshell> fac(15)
$53 ==> 1307674368000
```

Wir können dieses Programm von Hand ausführen, indem wir den Methodenaufruf für `fac` für einen konkreten Parameter `i` durch die für diesen Wert zutreffende Alternative

der Bedingungsabfrage ersetzen. Wir kennzeichnen einen solchen Ersetzungsschritt durch einen Pfeil →:

```
fac(4)
→4*fac(4-1)
→4*fac(3)
→4*(3*fac(3-1))
→4*(3*fac(2))
→4*(3*(2*fac(2-1)))
→4*(3*(2*fac(1)))
→4*(3*(2*(1*fac(1-1))))
→4*(3*(2*(1*fac(0))))
→4*(3*(2*(1*1)))
→4*(3*(2*1))
→4*(3*2)
→4*6
→24
```

3.4.1 Rekursion und Schleifen

Schleifen und Rekursion sind beide dazu geeignet, um Code-Teile wiederholt auszuführen. Man kann auch argumentieren, dass Schleifen ein zusätzliches Konstrukt von Programmiersprachen sind, mit denen sich Rekursion optimierter ausführen lassen können. Betrachten wir die for-Schleife, so lässt sich diese eigentlich immer direkt in eine rekursive Methode umschreiben.

Statt der Schleifenvariablen benötigt die rekursive Methode einen Parameter, der für die Steuerung der Rekursion in einer Abbruchbedingung verwendet wird. Der nächste Durchlauf wird durch den rekursiven Aufruf getätigt. Hierbei wird als neuer Wert der Parameter für die Rekursion übergeben. Eine if-Bedingung sorgt dafür, dass nur unter bestimmten Bedingungen ein weiterer rekursiver Aufruf kommt.

Eine Rekursion, die wie eine for-Schleife über einen Zahlenbereich geht, lässt sich damit wie folgt umsetzen:

```
jshell> void forRek(int from,int to){
   ...>   if (from<=to){
   ...>     System.out.println("hallo "+from );
   ...>     forRek(from+1,to);
   ...>   }
   ...> }
|  created method forRek(int,int)

jshell> forRek(3,7)
hallo 3
hallo 4
hallo 5
hallo 6
hallo 7
```

Wir nutzen aus, dass in Java Methoden überladen werden können. Zwei Methoden sind überladen, wenn sie denselben Methodennamen haben, aber unterschiedliche Parameter. Wir schreiben zunächst eine weitere überladene Version von `forRek`, die nur einen Parameter hat, der die obere Grenze der Rekursion angeben soll. Diese überladene Methode ruft direkt die erste Version auf, in dem der erste der beiden Parameter auf 1 gesetzt wird.

```
jshell> void forRek(int to){
   ...>   forRek(1,to);
   ...> }
|  created method forRek(int)
jshell> forRek(3)
hallo 1
hallo 2
hallo 3
```

Das lässt sich auch noch ein zweites Mal machen. Dieses Mal hat die überladene Version gar keinen Parameter.

```
jshell> void forRek(){
   ...>   forRek(10);
   ...> }
|  created method forRek()
```

Hier wird auch ein Standardwert genommen. Ruft man die Version ohne Parameter auf, so läuft die Rekursion 10-fach und es wird 10 Mal das Wort `hallo` auf der Kommandozeile ausgegeben.

Codeblöcke als Parameter in Java 8

Die rekursive Methode im obigen Abschnitt, die eine for-Schleife auf rekursive Weise simuliert ist nicht sehr allgemein. Sie ist nur in der Lage für eine bestimmte Anzahl den Befehl `System.out.println("hallo")` auszuführen. Mit der Version 1.8 von Java, die im Jahr 2014 eingeführt wurde, ist es möglich sein, Code-Fragmente als Parameter zu übergeben. Für solche Code-Fragmente wird dann ein Typ benutzt, der Typ `Runnable`.

Wir können dann die Methoden um einen weiteren Parameter erweitern. Dieser ist dann vom Typ `Runnable`.

```
jshell> void forRek(int from,int to,Runnable c){
   ...>   if (from<=to){
   ...>     c.run();
   ...>     forRek(from+1,to,c);
   ...>   }
   ...> }
|  created method forRek(int,int,Runnable)
```

Statt nun im eigentlichen Rumpf der rekursiven Methode, die Anweisung
`System.out.println("hallo")`
auszuführen, wird nun für das übergebene Objekt des Typs `Runnable` die Methode `run()` aufgerufen.

Es gibt eine Syntax geben, um Code-Fragmente als Blöcke an Methoden als Parameter zu übergeben. Dieses sind die sogenannten Lambda-Ausdrücke, die in diesem Fall `()->` gefolgt von einem in geschweiften Klammern eingeschlossenen Code-Block sind. Damit kann jetzt die rekursive Methode genutzt werden, um verschiedene Befehle mehrfach auszuführen.

```
jshell> forRek(5,9,()->{System.out.println("welt");})
welt
welt
welt
welt
welt
```

Die Änderungen, die mit Java 1.8 in die Sprache eingefügt wurden sind gewaltig und erlauben einen vollkommen neuen Programmierstil und eine Methodik, die stärker das funktionale Programmierparadigma verfolgt.

3.4.2 Einsatz von Rekursion

Wie wir gesehen haben, lassen sich mit Rekursion die gleichen Probleme lösen, wie mit der Iteration über Schleifen. Wann sind also besser Schleifen, wann rekursive Lösungen vorzuziehen. Wer die beiden konkurrierenden Versionen von `ForeverYoung` gestartet hat,

wird festgestellt haben, dass die rekursive Version relativ schnell abstürzt, während die iterative Version anscheinend endlos läuft.

Tatsächlich haben rekursive Methoden ein Problem: bei jedem Methodenaufruf ist auf einem Speicher abzulegen, von welcher Programmzeile mit was für konkreten Argumenten die Methode aufgerufen wurde. Diese Information wächst bei einer Rekursion schnell an, bis nicht mehr genügend Speicherplatz vorhanden ist, diese zu speichern. Dann stürzt das Programm ab. Daher ist für normale Wiederholungen immer die Schleife vorzuziehen. (Es sei denn man benutzt eine Programmiersprache, deren Compiler Rekursionen optimiert übersetzt.) Wir werden aber in Laufe des Moduls Datentypen kennenlernen, die in sich bereits rekursiv definiert sind und für die rekursive Lösungen adäquat sein können.

3.5 Reihungen (Arrays)

Java kennt, wie fast alle Programmiersprachen, ein weiteres Konzept von Sammlungen: Reihungen (eng. arrays). Reihungen stellen im Gegensatz zu Listen oder Mengen eine Menge von Daten gleichen Typs mit fester Anzahl dar. Jedes Element einer Reihung hat einen festen Index, über den es direkt angesprochen werden kann.

3.5.1 Deklaration von Reihungen

Eine Reihung hat einen festen Elementtyp.

Ein Reihungstyps wird deklariert, indem dem Elementtyp ein eckiges Klammernpaar nachgestellt wird, z.B. ist `String []` eine Reihung von Stringelementen. Die Elemente einer Reihung können sowohl von einem Objekttyp als auch von einem primitiven Typ sein, also gibt es auch den Typ `int []` oder z.B. `boolean []`.

Reihungen sind Objekte. Sie sind zuweisungskompatibel für Objektfelder, es lassen sich Typzusicherungen auf Reihungen durchführen und Reihungen haben ein Feld `length`vom Typ `int`, das die feste Länge einer Reihung angibt.

ObjectArray.java

```
class ObjectArray {
  public static void main(String [] args){
    Object as = args;
    System.out.println(((String [])as).length);
  }
}
```

3.5.2 Erzeugen von Reihungen

Es gibt zwei Verfahren, um Reihungen zu erzeugen: indem die Elemente der Reihung aufgezählt werden oder indem die Länge der Reihung angegeben wird. Eine Mischform,

in der sowohl Länge als auch die einzelnen Elemente angegeben werden, gibt es nicht.

Die einfachste Art, um eine Reihung zu erzeugen, ist, die Elemente aufzuzählen. Hierzu sind die Elemente in geschweiften Klammern mit Komma getrennt aufzuzählen:

```
jshell> String [] komponisten =
   ...>   {"carcassi","carulli","giuliani"
   ...>   ,"molino","monzino","paganini","sor"};
komponisten ==> String[7] { "carcassi", "carulli", "giuliani", "m ... zino"
, "paganini", "sor" }
```

Reihungen können nach der Anzahl der Elemente gefragt werden:

```
jshell> komponisten.length
$63 ==> 7
```

Auf Reihungen kann die Methode `toString()` aufgerufen werden. Allerdings hat diese keine sehr hilfreiche Umsetzung.

```
jshell> komponisten.toString()
$64 ==> "[Ljava.lang.String;@79079097"
```

Die jshell gibt wir einen Ausdruck, der eine Reihung ist, die Elemente alle an.

```
jshell> komponisten
komponisten ==> String[7] { "carcassi", "carulli", "giuliani", "molino", "mo
nzino", "paganini", "sor" }
```

Eine weitere Methode zur Erzeugung von Reihungen ist, noch nicht die einzelnen Elemente der Reihung anzugeben, sondern nur die Anzahl der Elemente:

```
jshell> int [] zahlenReihung = new int[10]
zahlenReihung ==> int[10] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
```

3.5.3 Zugriff auf Elemente

Die einzelnen Elemente einer Reihung können über einen Index angesprochen werden. Das erste Element einer Reihung hat den Index `0`, das letzte Element den Index `length-1`.

Als Syntax benutzt Java die auch aus anderen Programmiersprachen

bekannte Schreibweise mit eckigen Klammern:

```
jshell> komponisten[6]
$68 ==> "sor"
```

Typischer Weise wird mit einer `for`-Schleife über den Index einer Reihung iteriert. So läßt sich z.B. eine Methode,

die eine Stringdarstellung für Reihungen erzeugt, wie folgt schreiben:

ArrayToString.java

```
public class ArrayToString{
  static public String arrayToString(Object[] obja){
    StringBuffer result = new StringBuffer("{");

    for (int i=0;i<obja.length;i=i+1){
      if (i>0) result.append(",");
      result.append(obja[i].toString());
    }

    result.append("}");
    return result.toString();
  }
}
```

3.5.4 Ändern von Elementen

Eine Reihung kann als ein Komplex von vielen einzelnen Feldern gesehen werden. Die Felder haben keine eigenen Namen, sondern werden

über den Namen der Reihung zusammen mit ihrem Index angesprochen. Mit diesem Bild ergibt sich automatisch, wie nun einzelnen Reihungselementen neue Objekte zugewiesen werden können:

```
String [] stra = {"hello","world"};
stra[0]="hallo";
stra[1]="welt";
```

3.5.5 Die für-alle-Schleife

Eine häufige Aufgabe ist, für alle Elemente einer Reihung eine bestimmte Aktion durchzuführen. Wir haben schon die typische for-Schleife über den Index der Elemente gesehen. Java bietet aber für den Zweck, um über alle Elemente eines Array (oder sonstigen Sammlung) zu iterieren eine spezielle Variante der for-Schleife an, die sogenannte for-each Schleife.

```
jshell> for (var komponist:komponisten){
   ...>   System.out.println(komponist.toUpperCase());
   ...> }
CARCASSI
CARULLI
GIULIANI
MOLINO
MONZINO
PAGANINI
SOR
```

4 Weiterführende Konzepte der Objektorientierung

4.1 Vererbung

Eines der grundlegendsten Ziele der objektorientierten Programmierung ist die Möglichkeit, bestehende Programme um neue Funktionalität erweitern zu können. Hierzu bedient man sich der Vererbung. Bei der Definition einer neuen Klassen hat man die Möglichkeit, anzugeben, dass diese Klasse alle Eigenschaften von einer bestehenden Klasse erbt.

Wir haben in einer früheren Übungsaufgabe die Klasse `Person` geschrieben:

Person2.java

```
class Person2 {
  String name ;
  String address;

  Person2(String name, String address){
    this.name = name;
    this.address = address;
  }

  public String toString(){
    return name+", "+address;
  }
}
```

Wenn wir zusätzlich eine Klasse schreiben wollen, die nicht beliebige Personen speichern kann, sondern Studenten, die als zusätzliche Information noch eine Matrikelnummer haben, so stellen wir fest, dass wir wieder Felder für den Namen und die Adresse anlegen müssen; d.h. wir müssen die bereits in der Klasse `Person` zur Verfügung gestellte Funktionalität ein weiteres Mal schreiben:

StudentOhneVererbung.java

```
class StudentOhneVererbung {
  String name ;
  String address;
  int matrikelNummer;

  StudentOhneVererbung(String name, String address,int nr){
    this.name = name;
    this.address = address;
    matrikelNummer = nr;
  }

  public String toString(){
    return    name + ", " + address
            + " Matrikel-Nr.: " + matrikelNummer;
  }
}
```

Mit dem Prinzip der Vererbung wird es ermöglicht, diese Verdoppelung des Codes, der bereits für die Klasse `Person` geschrieben wurde, zu umgehen.

Wir werden in diesem Kapitel schrittweise eine Klasse `Student` entwickeln, die die Eigenschaften erbt, die wir in der Klasse `Person` bereits definiert haben.

Zunächst schreibt man in der Klassendeklaration der Klasse `Student`, dass deren Objekte alle Eigenschaften der Klasse `Person` erben. Hierzu wird das Schlüsselwort `extends` verwendet:

Student.java

```
class Student extends Person2 {
```

Mit dieser `extends`-Klausel wird angegeben, dass die Klasse von einer anderen Klasse abgeleitet wird und damit deren Eigenschaften erbt. Jetzt brauchen die Eigenschaften, die schon in der Klasse `Person` definiert wurden, nicht mehr neu definiert zu werden.

Mit der Vererbung steht ein Mechanismus zur Verfügung, der zwei primäre Anwendungen hat:

- Erweitern: zu den Eigenschaften der Oberklasse werden weitere Eigenschaften hinzugefügt. Im Beispiel der Studentenklasse soll das Feld `matrikelNummer` hinzugefügt werden.
- Verändern: eine Eigenschaft der Oberklasse wird umdefiniert. Im Beispiel der Studentenklasse soll die Methode `toString` der Oberklasse in ihrer Funktionalität verändert werden.

Es gibt in Java für eine Klasse immer nur genau eine direkte Oberklasse. Eine sogenannte

multiple Erbung ist in Java nicht möglich.[1] Es gibt immer maximal eine `extends`-Klausel in einer Klassendefinition.

4.1.1 Hinzufügen neuer Eigenschaften

Unser erstes Ziel der Vererbung war, eine bestehende Klasse um neue Eigenschaften zu erweitern. Hierzu können wir jetzt einfach mit der `extends`-Klausel angeben, dass wir die Eigenschaften einer Klasse erben. Die Eigenschaften, die wir zusätzlich haben wollen, lassen sich schließlich wie gewohnt deklarieren:

Student.java

```
  int matrikelNummer;
```

Hiermit haben wir eine Klasse geschrieben, die drei Felder hat: `name` und `adresse`, die von der Klasse `Person` geerbt werden und zusätzlich das Feld `matrikelNummer`. Diese drei Felder können für Objekte der Klasse `Student` in gleicher Weise benutzt werden:

Student.java

```
String writeAllFields(Student s){
   return s.name+" "+s.address+" "+s.matrikelNummer;
}
```

Ebenso so wie Felder lassen sich Methoden hinzufügen. Z.B. eine Methode, die die Matrikelnummer als Rückgabewert hat:

Student.java

```
int getMatrikelNummer(){
  return matrikelNummer;
}
```

4.1.2 Überschreiben bestehender Eigenschaften

Unser zweites Ziel ist, durch Vererbung eine Methode in ihrem Verhalten zu verändern. In unserem Beispiel soll die Methode `toString` der Klasse `Person` für Studentenobjekte so geändert werden, dass das Ergebnis auch die Matrikelnummer enthält. Hierzu können wir die entsprechende Methode in der Klasse `Student` einfach neu schreiben:

[1] Dieses ist z.B. in C++ möglich.

Student.java

```
@Override public String toString(){
  return    name + ", " + address
          + " Matrikel-Nr.: " + matrikelNummer;
}
```

Obwohl Objekte der Klasse `Student` auch Objekte der Klasse `Person` sind, benutzen sie nicht die Methode `toString` der Klasse `Person`, sondern die neu definierte Version aus der Klasse `Student`.

Um eine Methode zu überschreiben, muss sie dieselbe Signatur bekommen, die sie in der Oberklasse hat.

Um im Quelltext deutlich zu machen, dass man eine geerbte Methode überschreibt, kann man der Methode die Annotation `@Override` hinzufügen. Dieses hat keine Auswirkung auf das Laufzeitverhalten des Programms. Es bewirkt aber, dass der Kompilator überprüft, ob tatsächlich eine geerbte Methode neu definiert wird, und gegebenenfalls einen Fehler meldet, wenn dieses nicht der Fall ist.

Merken Sie sich:

Überschreiben (eng: override) bedeutet, dass eine aus einer Oberklasse geerbte Methoden in dieser Klasse neu definiert wird. Überladen (eng.: overload) bedeutet, dass eine Methode mit gleichen Namen in der derselben Klasse mehrfach existiert, aber unterschiedliche Parametertyoen hat.

4.1.3 Konstruktion

Um für eine Klasse konkrete Objekte zu konstruieren, braucht die Klasse entsprechende Konstruktoren. In unserem Beispiel soll jedes Objekt der Klasse `Student` auch ein Objekt der Klasse `Person` sein. Daraus folgt, dass, um ein Objekt der Klasse `Student` zu erzeugen, es auch notwendig ist, ein Objekt der Klasse `Person` zu erzeugen. Wenn wir also einen Konstruktor für `Student` schreiben, sollten wir sicherstellen, dass mit diesem auch ein gültiges Objekt der Klasse `Person` erzeugt wird. Hierzu kann man den Konstruktor der Oberklasse aufrufen. Dieses geschieht mit dem Schlüsselwort `super`. `super` ruft den Konstruktor der Oberklasse auf:

Student.java

```
  Student(String name,String adresse,int nr){
    super(name,adresse);
    matrikelNummer = nr;
  }
}
```

In unserem Beispiel bekommt der Konstruktor der

Klasse `Student` alle Daten, die benötigt werden, um ein Personenobjekt und ein Studentenobjekt zu erzeugen. Als erstes wird im Rumpf des Studentenkonstruktors der Konstruktor der Klasse `Person` aufgerufen. Anschließend wird das zusätzliche Feld der Klasse `Student` mit entsprechenden Daten initialisiert.

Ein Objekt der Klasse `Student` kann wie gewohnt konstruiert werden:

TestStudent.java

```
class TestStudent {
  public static void main(String [] args){
    Student s
     = new Student("Martin Müller","Hauptstraße 2",755423);
    System.out.println(s);
  }
}
```

4.1.4 Zuweisungskompatibilität

Objekte einer Klasse sind auch ebenso Objekte ihrer Oberklasse. Daher können sie benutzt werden wie die Objekte ihrer Oberklasse, insbesondere bei einer Zuweisung. Da in unserem Beispiel die Objekte der Klasse `Student` auch Objekte der Klasse `Person` sind, dürfen diese auch Feldern des Typs `Person` zugewiesen werden:

TestStudent1.java

```
class TestStudent1{
  public static void main(String [] args){
    Person2 p
     = new Student("Martin Müller","Hauptstraße",7463456);
  }
}
```

Alle Studenten sind auch Personen.

Hingegen die andere Richtung ist nicht möglich: nicht alle Personen sind Studenten. Folgendes Programm wird von Java mit einem Fehler zurückgewiesen:

Fehlerhafter Code!

```
class StudentError1{
  public static void main(String [] args){
    Student s
      = new Person2("Martin Müller","Hauptstraße");
  }
}
```

Die Kompilierung dieser Klasse führt zu folgender Fehlermeldung:

```
StudentError1.java:3: incompatible types
found   : Person
required: Student
    Student s = new Person2("Martin Müller","Hauptstraße");
                ^
1 error
```

Java weist diese Klasse zurück, weil eine Person nicht ein Student ist.

Gleiches gilt für den Typ von Methodenparametern. Wenn die Methode einen Parameter vom Typ `Person` verlangt, so kann man ihm auch Objekte eines spezielleren Typs geben, in unserem Fall der Klasse `Student`.

TestStudent2.java

```
class TestStudent2 {

  static void printPerson(Person2 p){
    System.out.println(p.toString());
  }

  public static void main(String [] args){
     Student s
       = new Student("Martin Müller","Hauptstraße",754545);
     printPerson(s);
  }
}
```

Der umgekehrte Fall ist wiederum nicht möglich. Methoden, die als Parameter Objekte der Klasse `Student` verlangen, dürfen nicht mit Objekten einer allgemeineren Klasse aufgerufen werden:

Fehlerhafter Code!

```
class StudentError2{
  static void printStudent(Student s){
    System.out.println(s.toString());
  }

  public static void main(String [] args){
     Person2 p = new Person2("Martin Müller","Hauptstraße");
     printStudent(p);
  }
}
```

Auch hier führt die Kompilierung zu einer entsprechenden Fehlermeldung:

```
StudentError2.java:9: printStudent(Student) in StudentError2
                      cannot be applied to (Person2)
     printStudent(p);
     ^
1 error
```

4.1.5 Späte Bindung (late binding)

Wir haben gesehen, dass wir Methoden überschreiben können. Interessant ist, wann welche Methode ausgeführt wird. In unserem Beispiel gibt es je eine Methode `toString` in der Oberklasse `Person` als auch in der Unterklasse `Student`.

Welche dieser zwei Methoden wird wann ausgeführt? Wir können dieser Frage experimentell nachgehen.

Hierzu definieren wir ein Klasse `A` mit nur einer Methode `info()`.

```
jshell> class A{
   ...>   String info(){return "Klasse A";}
   ...> }
|  created class A
```

Zu dieser Klasse definieren wir eine Unterklasse, die die Methode `info()` überschreibt.

```
jshell> class B extends A{
   ...>    @Override String info(){return "Klasse B";}
   ...> }
|  created class B
```

Erzeugen wir ein Objekt der Klasse `A` und führen die Methode `info()` aus, so wird die Methode aus der Klasse `A` ausgeführt.

```
jshell> A a = new A()
a ==> A@3b95a09c

jshell> a.info()
$72 ==> "Klasse A"
```

Erzeugen wir ein Objekt der Klasse `B` und führen die Methode `info()` aus, so wird die Methode aus der Klasse `B` ausgeführt.

```
jshell> B b = new B()
b ==> B@3d82c5f3

jshell> b.info()
$74 ==> "Klasse B"
```

Schließlich erzeugen wir ein Objekt der Klasse `B` und speichern es in einer Variablen des Typs der Klasse `A`.

```
jshell> A c = new B()
c ==> B@66cd51c3

jshell> /var c
|    A c = B@66cd51c3
```

Führen die Methode `info()` für das Objekt der Variablen `c` des Typs `A`aus, so wird die Methode aus der Klasse `B` ausgeführt, weil das dort gespeicherte Objekt von der Klasse `B` erzeugt wird.

```
jshell> c.info()
$76 ==> "Klasse B"
```

Das Objekt vergisst also nicht, von welcher Klasse es ist.

Dieses in Java realisierte Prinzip wird als späte Bindung (eng. late binding) bezeichnet.[2]

Zurück zu den Personen und Studenten. Auch hier wird ein Studentenobjekt nie vergessen, dass es von der Klasse `Student` ist, obwohl wir es nur über eine Variable des Typs `Person` ansprechen.

TestLateBinding.java

```
class TestLateBinding {
   public static void main(String [] args){
    Student s = new Student("Martin Müller","Hauptstraße",756456);
    Person2 p1 = new Person2("Harald Schmidt","Marktplatz");

    System.out.println(s.toString());
    System.out.println(p1.toString());

    Person2 p2 = new Student("Martin Müller","Hauptstraße",756456);
    System.out.println(p2.toString());
  }
}
```

Dieses Programm erzeugt folgende Ausgabe:

[2] Achtung: *late binding* funktioniert in Java nur bei Methoden, nicht bei Feldern.

```
sep@swe10:~/fh/> java TestLateBinding
Martin Müller, Hauptstraße Matrikel-Nr.: 756456
Harald Schmidt, Marktplatz
Martin Müller, Hauptstraße Matrikel-Nr.: 756456
```

Die ersten beiden Ausgaben entsprechen sicherlich den Erwartungen: es wird eine Student und anschließend eine Person ausgegeben. Die dritte Ausgabe ist interessant. Obwohl der Befehl:

```
System.out.println(p2.toString());
```

die Methode `toString` auf einem Feld vom Typ `Person` ausführt, wird die Methode `toString` aus der Klasse `Student` ausgeführt. Dieser Effekt entsteht, weil das Objekt, das im Feld `p2` gespeichert wurde, als Student und nicht als Person erzeugt wurde. Die Idee der Objektorientierung ist, dass die Objekte die Methoden in sich enthalten. In unserem Fall enthält das Objekt im Feld `p2` seine eigene `toString`-Methode. Diese wird ausgeführt. Der Ausdruck `p2.toString()` ist also zu lesen als: Objekt, das in Feld `p2` gespeichert ist, führe bitte deine Methode `toString` aus.

Da dieses Objekt, auch wenn wir es dem Feld nicht ansehen, ein Objekt der Klasse `Student` ist, führt es die entsprechende Methode der Klasse `Student` und nicht der Klasse `Person` aus.

Beispiel zum Nutzen der späten Bindung

Vererbung, Überschreiben von Methoden und späte Bindung, können helfen, Software gut in unabhängige Komponenten zu teilen und in getrennten Teams zu arbeiten. Stellen Sie sich vor, dass Sie ein Team haben, das geübt darin ist, braphische Benutzeroberflächen zu entwickeln. Sie sind nicht in diesem Team, aber das Team schreibt die folgende Klassen `Dialogue`, die Sie selbst nicht verstehen.

Dialogue.java

```
import javax.swing.*;
import java.awt.*;
class Dialogue extends JFrame{
  final JTextField inputField  = new JTextField(20) ;
  final JTextField outputField = new JTextField(20) ;
  final JPanel p = new JPanel();
  Dialogue(ButtonLogic l){
    JButton button=new JButton(l.getDescription());
    button.addActionListener
        (ev -> outputField.setText(l.eval(inputField.getText().trim())));
    p.setLayout(new BorderLayout());
    p.add(inputField,BorderLayout.NORTH);
    p.add(button,BorderLayout.CENTER);
    p.add(outputField,BorderLayout.SOUTH);
    add(p);
    pack();
    setVisible(true);
  }
}
```

Die dort im Konstruktor verwendete Klasse `ButtonLogic` ist sehr einfach und schnell verstanden.

ButtonLogic.java

```
class ButtonLogic {
  String getDescription(){return "Knopf Drücken";}
  String eval(String x){return x.toUpperCase();}
}
```

Sie können die beiden Klassen einmal in der jshell ausprobieren:

```
jshell> /open ButtonLogic.java

jshell> /open Dialogue.java

jshell> var bl = new ButtonLogic()
bl ==> ButtonLogic@150c158

jshell> var dia = new Dialogue(bl)
dia ==> Dialogue[frame1,0,24,224x91,layout=java.awt.Borde ... tPaneCheckingE

jshell>
```

Es öffnet sich ein Fenster. Sie können einen Text eingeben, und wenn Sie den Knopf drücken, wird der Text in Großbuchstaben angezeigt. Wenn Sie jetzt eine Unterklasse von `ButtonLogic` schreiben, zum Beispiel zum verstehen von römischen Zahlen:

RomanLogic.java

```
class RomanLogic extends ButtonLogic {
  @Override String eval(String str){
    var r=0;
    var last = 1000;
    for (var c:str.toUpperCase().toCharArray()){
      var v = getValue(c);
      if (v>last) r -= 2*last;
      r += v;
      last = v;
    }
    return r+"";
  }
```

Eine kleine Hilfsmethode, die dabei verwendet wurde, berechnet für jedes römische Zahlensymbol den entsprechenden Zahlenwert:

RomanLogic.java

```
  int getValue(char c){
    switch(c){
      case 'I': return 1;
      case 'V': return 5;
      case 'X': return 10;
      case 'L': return 50;
      case 'C': return 100;
      case 'D': return 500;
      case 'M': return 1000;
    }
    return 0;
  }
}
```

Nun können Sie die GUI-Klasse `Dialogue` mit einem Objekt Ihrer neuen Klasse `RomanLogic` aufrufen und erhalten eine GUI-Anwendung, in der die eingegebenen römischen Zahlen in arabischen Zahlen angezeigt werden. Hierzu mussten Sie die eigentliche GUI-Klasse nicht in ihren Interna kennen. Sie konnten durch Überschreiben der Methoden einer Klasse, dem GUI eine eigene Anwendungslogik geben.

```
jshell> /open RomanLogic.java

jshell> new Dialogue(new RomanLogic())
```

4.1.6 Zugriff auf Methoden der Oberklasse

Vergleichen wir die Methoden `toString` der Klassen `Person` und `Student`, so sehen wir, daß in der Klasse `Student` Code der Oberklasse verdoppelt wurde:

```
public String toString(){
  return    name + ", " + address
          + " Matrikel-Nr.: " + matrikelNummer;
}
```

Der Ausdruck

```
name + ", " + address
```

wiederholt die Berechnung der `toString`-Methode aus der Klasse `Person`. Es wäre schön, wenn an dieser Stelle die entsprechende Methode aus der Oberklasse benutzt werden könnte. Auch dieses ist in Java möglich. Ähnlich, wie der Konstruktor der Oberklasse explizit aufgerufen werden kann, können auch Methoden der Oberklasse explizit aufgerufen werden. Auch in diesem Fall ist das Schlüsselwort `super` zu benutzen, allerdings nicht in der Weise, als sei `super` eine Methode, sondern als sei es ein Feld, das ein Objekt enthält, also ohne Argumentklammern. Dieses Feld erlaubt es, direkt auf die Eigenschaften der Oberklasse zuzugreifen. Somit läßt sich die `toString`-Methode der Klasse `Student` auch wie folgt schreiben:

```
public String toString(){
  return      //call toString of super class
              super.toString()
              //add the Matrikelnummer
          + " Matrikel-Nr.: " + matrikelNummer;
}
```

4.1.7 Die Klasse Object

Eine berechtigte Frage ist, welche Klasse die Oberklasse für eine Klasse ist, wenn es keine `extends`-Klausel gibt. Bisher haben wir nie eine entsprechende Oberklasse angegeben.

Java hat in diesem Fall eine Standardklasse: `Object`. Wenn nicht explizit eine Oberklasse angegeben wird, so ist die Klasse `Object` die direkte Oberklasse. Weil die `extends`-Relation transitiv ist, ist schließlich jede Klasse eine Unterklasse der Klasse `Object`. Insgesamt bilden alle Klassen, die in Java existieren, eine Baumstruktur, deren Wurzel die Klasse `Object` ist.

Es bewahrheitet sich die Vermutung über objektorientierte Programmierung, dass alles

als Objekt betrachtet wird.[3] Es folgt insbesondere, dass jedes Objekt die Eigenschaften hat, die in der Klasse `Object` definiert wurden.

Die Eigenschaften, die alle Objekte haben, weil sie in der Klasse `Object` definiert sind, sind äußerst allgemein. Sobald wir von einem Object nur noch wissen, dass es vom Typ `Object` ist, können wir kaum noch spezifische Dinge mit ihm anfangen.

Die Methode `toString`

Ein Blick in die Java API Documentation zeigt, dass zu diesen Eigenschaften auch die Methode `toString` gehört, wie wir sie bereits einige mal geschrieben haben. Jetzt erkennen wir, dass wir diese Methode dann überschrieben haben. Auch wenn wir für eine selbstgeschriebene Klasse die Methode `toString` nicht definiert haben, existiert eine solche Methode. Allerdings ist deren Verhalten selten ein für unsere Zwecke geeignetes. Wir haben das schon gesehen, wenn wir mit `println` Objekte von Klassen, in denen die Methode `toString` nicht überschrieben wurde, ausgegeben haben, oder ein solches Objekt Ergebnis eines Ausdrucks in der jshell war.

Laut Dokumentation ist die Methode `toString` in der Klasse Objekt wie folgt definiert:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Es werden zwei weitere Methoden, die bereits in der Klasse `Object` definiert sind, aufgerufen: `getClass` und `hashCode`.

Die Methode `equals`

Eine weitere Methode, die in der Klasse `Object` definiert ist, ist die Methode `equals`. Sie hat folgende Signatur:

```
public boolean equals(Object other)
```

Wenn man diese Methode überschreibt, so kann definiert werden, wann zwei Objekte einer Klasse als gleich angesehen werden sollen. Für Personen würden wir gerne definieren, dass zwei Objekte dieser Klasse gleich sind, wenn sie ein und denselben Namen und ein und dieselbe Adresse haben. Mit unseren derzeitigen Mitteln läßt sich dieses leider nicht ausdrücken. Wir würden gerne die `equals`-Methode wie folgt überschreiben:

[3] Java kennt acht eingebaute primitive Typen für Zahlen, Wahrheitswerte und Buchstaben. Diese sind zwar keine Objekte, werden notfalls von Java aber in entsprechende Objektklassen automatisch konvertiert.

Fehlerhafter Code!

```
public boolean equals(Object other){
  return        this.name.equals(other.name)
             && this.adresse.equals(other.adresse);
}
```

Dieses ist aber nicht möglich, weil für das Objekt `other`, von dem wir nur wissen, dass es vom Typ `Object` ist, keine Felder `name` und `adresse` existieren.

Um dieses Problem zu umgehen, sind Konstrukte notwendig, die von allgemeineren Typen wieder zu spezielleren Typen führen. Ein solches Konstrukt lernen wir in den folgenden Abschnitten kennen.

Test auf Klassenzugehörigkeit Wie wir oben gesehen haben, können wir zu wenige Informationen über den Typen eines Objektes haben. Objekte wissen aber selbst, von welcher Klasse sie einmal erzeugt wurden. Java stellt einen binären Operator zur Verfügung, der erlaubt, abzufragen, ob ein Objekt zu einer Klasse gehört. Dieser Operator heißt `instanceof`. Er hat links ein Objekt und rechts einen Klassennamen. Das Ergebnis ist ein bool'scher Wert, der genau dann wahr ist, wenn das Objekt eine Instanz der Klasse ist.

```
jshell> Object p1 = new Person2("Strindberg","Skandinavien");
   ...> Object p2 = new Student("Ibsen","Skandinavien",789565);
p1 ==> Strindberg, Skandinavien
p2 ==> Ibsen, Skandinavien Matrikel-Nr.: 789565

jshell> p1 instanceof Student
$17 ==> false

jshell> p2 instanceof Student
$18 ==> true

jshell> p1 instanceof Person2
$19 ==> true

jshell> p2 instanceof Person2
$20 ==> true
```

An der Ausgabe dieses Programms kann man erkennen, dass ein `instanceof`-Ausdruck wahr wird, wenn das Objekt ein Objekt der Klasse oder aber einer Unterklasse der Klasse des zweiten Operanden ist.

Damit gilt natürlich für jedes Objekt `o`, egal von welcher Klasse es einmal erzeugt wurde, dass der Ausdruck `o instanceof Object` zu `true` auswertet.

```
jshell> p1 instanceof Object
$21 ==> true
```

Das Klassenobjekt Es gibt eine zweite Möglichkeit, um zu testen, ob ein Objekt von einer bestimmten Klasse ist. In der Klasse `Object` existiert die Methode `getClass()`. Diese gibt ein Objekt zurück, dass die Klasse beschreibt, von der das Objekt erzeugt wurde.

```
jshell> p1.getClass()
$9 ==> class Person2

jshell> p2.getClass()
$10 ==> class Student
```

Obwohl die Variablen `p1` und `p2` vom Typ `Object` sind, bekommen wir mit der Methode `getClass()` genau gesagt, von welcher Klasse die Objekte in den Variablen erzeugt wurden.

Dieses lässt sich jetzt auch nutzen, ob ein Objekt von einer bestimmten Klasse ist. In jeder Klasse gibt es nämlich das statische Feld `class`. Dieses gibt ein Objekt zurück, dass die Klasse beschreibt. Jetzt können wir prüfen, ob dieses identisch ist mit der Klasse, von der unser Objekt ist.

```
jshell> p2.getClass()==Person2.class
$11 ==> false

jshell> p1.getClass()==Person2.class
$12 ==> true

jshell> p2.getClass()==Student.class
$13 ==> true

jshell> p1.getClass()==Student.class
$14 ==> false
```

Auf diesen Weg können wir also nicht prüfen, ob ein Objekt von einer Klasse oder einer der Unterklassen ist, sondern exakt von dieser Klasse erzeugt wurde.

Typzusicherung In den letzten Abschnitten haben wir zwei Möglichkeiten kennengelernt, zu fragen, ob ein Objekt zu einer bestimmten Klasse gehört. Um ein Objekt dann auch wieder so benutzen zu können, dass es zu dieser Klasse gehört, müssen wir diesem Objekt diesen Typ erst wieder zusichern. Im obigen Beispiel haben wir zwar erfragen können, dass das in Feld `p2` gespeicherte Objekt nicht nur eine Person, sondern ein Student ist; trotzdem können wir noch nicht `p2` nach seiner Matrikelnummer fragen. Hierzu müssen wir erst zusichern, dass das Objekt den Typ `Student` hat.

Eine Typzusicherung in Java wird gemacht, indem dem entsprechenden Objekt in Klammer der Typ vorangestellt wird, den wir ihm zusichern wollen:

```
jshell> Student s = (Student)p2
s ==> Ibsen, Skandinavien Matrikel-Nr.: 789565

jshell> s.matrikelNummer
$23 ==> 789565
```

Die Zeile `s = (Student)p2` sichert erst dem Objekt im Feld `p2` zu, dass es ein Objekt des Typs `Student` ist, so dass es dann als Student benutzt werden kann. Wir haben den Weg zurück vom Allgemeinen ins Spezifischere gefunden. Allerdings ist dieser Weg gefährlich. Eine Typzusicherung kann fehlschlagen:

```
jshell> Student s2 = (Student)p1
|  Exception java.lang.ClassCastException
```

Dieses Programm macht eine Typzusicherung des Typs `Student` auf ein Objekt, das nicht von diesem Typ ist. Es kommt in diesem Fall zu einen Laufzeitfehler:

Will man solche Laufzeitfehler verhindern, so ist man auf der sicheren Seite, wenn eine Typzusicherung nur dann gemacht wird, nachdem man sich mit einem `instanceof`-Ausdruck davon überzeugt hat, dass das Objekt wirklich von dem Typ ist, den man ihm zusichern will.

Mit den jetzt vorgestellten Konstrukten können wir eine Lösung der Methode `equals` für die Klasse `Person2` mit der erwarteten Funktionalität schreiben:

```
@Override public boolean equals(Object other){
  if (other==null) return false;
  if (!(other instanceof Person2)) return false;
  Person2 that = (Person2) other;
  if (!this.name.equals(that.name)) return false
  if (!this.adresse.equals(that.adresse)) return false;
  return true;
}
```

Nur, wenn das zu vergleichende Objekt auch vom Typ `Person2` ist und den gleichen Namen und die gleiche Adresse hat, dann sind zwei Personen gleich.

Wir verwenden hier eine Technik, die ich gerne als Sherlock Holmes Prinzip bezeichne. Schließe alle negativen Fälle aus. In diesem Fall alle die Fälle, in denen die beiden Objekte nicht gleich sind. Wenn die alle ausgeschlossen wurde, muss das Ergebnis wohl wahr sein.

Dasselbe und das Gleiche `public boolean equals(Object obj)` ist die Methode zum Testen auf die Gleichheit von zwei Objekten. Dabei ist die Gleichheit nicht zu verwechseln

mit der Identität, die mit dem Operator == getestet wird. Der Unterschied zwischen Identität und Gleichheit läßt sich sehr schön an Strings demonstrieren.

Wir erzeugen zunächst zwei Strings, die aus denselben Zeichen bestehen.

```
jshell> var x = "hallo".toUpperCase()
x ==> "HALLO"

jshell> var y = "hallo".toUpperCase()
y ==> "HALLO"
```

Wenn wir ein Objekt mit sich vergleichen, ergibt das wahr. Ebenso ergibt es wahr, wenn zwei Strings mit gleichen Zeichen verglichen werden:

```
jshell> x.equals(x)
$27 ==> true

jshell> x.equals(y)
$28 ==> true

jshell> y.equals(x)
$29 ==> true
```

Anders verhält es ich mit dem Operator == für Objekte. Dieser prüft nicht auf inhaltliche Gleichheit, sondern ob es sich um ein und dasselbe Objekt an der selben Stelle im Speicher handelt.

```
jshell> x==x
$30 ==> true

jshell> x==y
$31 ==> false
```

Obwohl die beiden Objekte x und y die gleichen Texte darstellen, sind es zwei unabhängige Objekte; sie sind nicht identisch, aber gleich.

Auch in der deutschen Sprache gibt es die subtile Unterscheidung zwischen der Identität und einer Gleichheit. Meinen wir eine Gleichheit benützen wir den Ausdruck ›das Gleiche‹, wollen wir die Identität von einem Objekt ausdrücken, so sprechen wir von ›dasselbe‹. Man kann sich das an einer Alltagssituation gut verdeutlichen. Wenn Sie in einem Restaurant dem Kellner sagen, Sie wollen das gleiche Gericht, wie die Dame am Nebentisch, so wird der Kellner in die Küche gehen und dort einen frischen Teller mit eben den Gericht, dass die Dame am Nebentisch bekommen hat, zubereiten lassen. Es gibt also zwei Teller mit zweimal dem gleichen Gericht. Verlangen Sie hingegen dasselbe, wie die Dame am Nebentisch, so muss der Kellner streng genommen der Dame den Teller wegnehmen und Ihnen hinstellen. Es gibt also nur ein einziges Objekt. das Gleiche heißt in Java »equals«, dasselbe bedeutet in Java ==. Nur bei primitiven Typen gibt es

keinen Unterschied. Dort stellen die Daten nämlich gar keine Objekte dar. Die Klasse `String` hingegen ist kein primitiver Typ und deshalb ist in der Regel zum Vergleichen von zwei String-Objekten die Methode `equals` aufzurufen und nicht der Operator `==` zu verwenden.

Sofern die Methode `equals` für eine Klasse nicht überschrieben wird, wird die entsprechende Methode aus der Klasse `Object` benutzt. Diese überprüft aber keine inhaltliche Gleichheit. Es ist also zu empfehlen, die Methode `equals` für alle eigenen Klassen, die zur Datenhaltung geschrieben wurden, zu überschreiben. Dabei sollte die Methode immer folgender Spezifikation genügen:

- **Reflexivität**: es sollte immer gelten: `x.equals(x)`
- **Symmetrie**: wenn `x.equals(y)` dann auch `y.equals(x)`
- **Transitivität**: wenn `x.equals(y)` und `y.equals(z)` dann gilt auch `x.equals(z)`
- **Konsistenz**: wiederholte Aufrufe von `equals` auf dieselben Objekte liefern dasselbe Ergebnis, sofern die Objekte nicht verändert wurden.
- nichts gleicht `null`: `x.equals(null)` ist immer falsch.

4.2 Pakete

Java bietet die Möglichkeit, Klassen in Paketen zu sammeln. Die Klassen eines Paketes bilden zumeist eine funktional logische Einheit. Pakete sind hierarchisch strukturiert, d.h. Pakete können Unterpakete haben. Damit entsprechen Pakete Ordnern im Dateisystem. Pakete ermöglichen verschiedene Klassen gleichen Namens, die unterschiedlichen Paketen zugeordnet sind.

4.2.1 Paketdeklaration

Zu Beginn einer Klassendefinition kann eine Paketzugehörigkeit für die Klasse definiert werden. Dieses geschieht mit dem Schlüsselwort `package` gefolgt von dem gewünschten Paket. Die Paketdeklaration schließt mit einem Semikolon.

Folgende Klasse definiert sie dem Paket `testPackage` zugehörig:

MyClass.java

```
package testPackage;
class MyClass {
}
```

Unterpakete werden von Paketen mit Punkten abgetrennt. Folgende Klasse wird dem Paket `testPackages` zugeordnet, das ein Unterpaket des Pakets `panitz` ist, welches wiederum ein Unterpaket des Pakets `name` ist:

TestPaket.java

```
package name.panitz.testPackages;
class TestPaket {
  public static void main(String [] args){
    System.out.println("hello from package \'testpackages\'");
  }
}
```

Paketnamen werden per Konvention in lateinischer Schrift immer mit Kleinbuchstaben als erstem Buchstaben geschrieben.

Wie man sieht, kann man eine weltweite Eindeutigkeit seiner Paketnamen erreichen, wenn man die eigene Webadresse hierzu benutzt.[4] Dabei wird die Webadresse rückwärts verwendet.

Paketname und Klassenname zusammen identifizieren eine Klasse eindeutig. Jeder Programmierer schreibt sicherlich eine Vielzahl von Klassen `Test`, es gibt aber in der Regel nur einen Programmierer, der diese für das Paket `name.panitz.testPackages` schreibt. Paket- und Klassenname zusammen durch einen Punkt getrennt werden der *vollqualifizierte Name* der Klasse genannt, im obigen Beispiel ist entsprechend der vollqualifizierte Name:
`name.panitz.testPackages.Test`

Der Name einer Klasse ohne die Paketnennung heißt unqualifiziert.

4.2.2 Übersetzen von Paketen

Bei größeren Projekten ist es zu empfehlen, die Quelltexte der Javaklassen in Dateien zu speichern, die im Dateisystem in einer Ordnerstruktur, die der Paketstruktur entspricht, liegen. Dieses ist allerdings nicht unbedingt zwingend notwendig. Hingegen zwingend notwendig ist es, die erzeugten Klassendateien in Ordnern entsprechend der Paketstruktur zu speichern.

Der Javainterpreter `java` sucht nach Klassen in den Ordnern entsprechend ihrer Paketstruktur. `java` erwartet also, dass die obige Klasse `Test` in einem Ordner `testPackages` steht, der ein Unterordner des Ordners `panitz` ist, der ein Unterordner des Ordners `tfhberlin` ist. usw. `java` sucht diese Ordnerstruktur von einem oder mehreren Startordnern ausgehend. Die Startordner werden in einer Umgebungsvariablen `CLASSPATH` des Betriebssystems und über den Kommandozeilenparameter `-classpath` festgelegt.

Der Javaübersetzer `javac` hat eine Option, mit der gesteuert

wird, dass `javac` für seine `.class`-Dateien die notwendige Ordnerstruktur erzeugt und die Klassen in die ihren Paketen entsprechenden Ordner schreibt. Die Option heißt `-d`.

[4] Leider ist es in Deutschland weit verbreitet, einen Bindestrich in Webadressen zu verwenden. Der Bindestrich ist leider eines der wenigen Zeichen, die Java in Klassen- und Paketnamen nicht zuläßt.

Dem `-d` ist nachgestellt, von welchem Startordner aus die Paketordner erzeugt werden sollen. Memotechnisch steht das `-d` für *destination*.

Wir können die obige Klasse z.B. übersetzen mit folgendem Befehl auf der Kommandozeile:
`javac -d . Test.java`

Damit wird ausgehend vom aktuellem Verzeichnis[5] ein Ordner `de` mit Unterordner `tfhberlin` etc. erzeugt.

4.2.3 Starten von Klassen in Paketen

Um Klassen vom Javainterpreter zu starten, reicht es nicht, ihren Namen anzugeben, sondern der vollqualifizierte Name ist anzugeben. Unsere obige kleine Testklasse wird also wie folgt gestartet:

```
sep@swe10:~/> java name.panitz.testPackages.Test
hello from package 'testpackages'
sep@swe10:~/>
```

Jetzt erkennt man auch, warum dem Javainterpreter nicht die Dateiendung `.class` mit angegeben wird. Der Punkt separiert Paket- und Klassennamen.

Aufmerksame Leser werden bemerkt haben, dass der Punkt in Java durchaus konsistent mit einer Bedeutung verwendet wird: hierzu lese man ihn als *'enthält ein'*. Der Ausdruck:
`name.panitz.testPackages.Test.main(args)`
liest sich so als: das Paket `de` enthält ein Unterpaket `tfhberlin`, das ein Unterpaket `panitz` enthält, das ein Unterpaket `testpackages` enthält, das eine Klasse `Test` enthält, die eine Methode `main` enthält.

4.2.4 Das Java Standardpaket

Die mit Java mitgelieferten Klassen sind auch in Paketen gruppiert. Die Standardklassen wie z.B. `String` und `System` und natürlich auch `Object` liegen im Java-Standardpaket `java.lang`. Java hat aber noch eine ganze Reihe weitere Pakete, so z.B. `java.util`, in dem sich Listenklassen befinden, `java.applet`, in dem Klassen zur Programmierung von Applets auf HTML-Seiten liegen, oder `java.io`, welches Klassen für Eingaben und Ausgaben enthält.

4.2.5 Benutzung von Klassen in anderen Paketen

Um Klassen benutzen zu können, die in anderen Paketen liegen, müssen diese eindeutig über ihr Paket identifiziert werden. Dieses kann dadurch geschehen, dass die Klassen

[5] Der Punkt steht in den meisten Betriebssystemen für den aktuellen Ordner, in dem gerade ein Befehl ausgeführt wird.

immer vollqualifiziert angegeben werden. Im folgenden Beispiel benutzen wir die Standardklasse `ArrayList`[6] aus dem Paket `java.util`.

TestArrayList.java

```
package name.panitz.utilTest;
class TestArrayList {
  public static void main(String [] args){
    java.util.ArrayList<String> xs = new java.util.ArrayList<String>();
    xs.add("friends");
    xs.add("romans");
    xs.add("countrymen");
    System.out.println(xs);
  }
}
```

Wie man sieht, ist der Klassenname auch beim Aufruf des Konstruktors vollqualifiziert anzugeben.

4.2.6 Importieren von Paketen und Klassen

Importieren von Klassen

Vollqualifizierte Namen können sehr lang werden. Wenn Klassen, die in einem anderen Paket als die eigene Klasse liegen, unqualifiziert benutzt werden sollen, dann kann dieses zuvor angegeben werden. Dieses geschieht zu Beginn einer Klasse in einer Importanweisung. Nur die Klassen aus dem Standardpaket `java.lang` brauchen nicht explizit durch eine Importanweisung bekannt gemacht zu werden.

Unsere Testklasse aus dem letzten Abschnitt kann mit Hilfe einer Importanweisung so geschrieben werden, dass die Klasse `ArrayList` unqualifiziert[7] benutzt werden kann:

[6] `ArrayList` ist eine generische Klasse, ein Konzept, das wir erst in einem späteren Kapitel kennenlernen werden.

[7] Aus historischen Gründen wird in diesem Kapitel als Beispiel bereits mit den generischen Klassen `ArrayList` und `Vector` ein Konzept benutzt, das erst im nächsten Kapitel erklärt wird.

TestImport.java

```
package name.panitz.utilTest;

import java.util.ArrayList;

class TestImport {
  public static void main(String [] args){
    ArrayList<String> xs = new ArrayList<String>();
    xs.add("friends");
    xs.add("romans");
    xs.add("countrymen");
    System.out.println(xs);
  }
}
```

Es können mehrere Importanweisungen in einer Klasse stehen. So können wir z.B. zusätzlich die Klasse `Vector` importieren:

TestImport2.java

```
package name.panitz.utilTest;

import java.util.ArrayList;
import java.util.Vector;

class TestImport2 {
  public static void main(String [] args){
    ArrayList<String> xs = new ArrayList<String>();
    xs.add("friends");
    xs.add("romans");
    xs.add("countrymen");
    System.out.println(xs);

    Vector<String> ys = new Vector<String>();
    ys.add("friends");
    ys.add("romans");
    ys.add("countrymen");
    System.out.println(ys);
  }
}
```

Importieren von Paketen

Wenn in einem Programm viele Klassen eines Paketes benutzt werden, so können mit einer Importanweisung auch alle Klassen dieses Paketes importiert werden. Hierzu gibt man in der Importanweisung einfach statt des Klassennamens ein `*` an.

TestImport3.java

```
package name.panitz.utilTest;

import java.util.*;

class TestImport3 {
  public static void main(String [] args){
    List<String> xs = new ArrayList<String>();
    xs.add("friends");
    System.out.println(xs);

    Vector<String> ys = new Vector<String>();
    ys.add("romans");
    System.out.println(ys);
  }
}
```

Ebenso wie mehrere Klassen können auch mehrere komplette Pakete importiert werden. Es können auch gemischt einzelne Klassen und ganze Pakete importiert werden.

4.2.7 Statische Imports

Statische Eigenschaften einer Klasse werden in Java dadurch angesprochen, dass dem Namen der Klasse mit Punkt getrennt die gewünschte Eigenschaft folgt. Werden in einer Klasse sehr oft statische Eigenschaften einer anderen Klasse benutzt, so ist der Code mit deren Klassennamen durchsetzt. Die Javaentwickler haben mit Java 1.5 ein Einsehen. Man kann jetzt für eine Klasse alle ihre statischen Eigenschaften importieren, so dass diese unqualifiziert benutzt werden kann. Die `import`-Anweisung sieht aus wie ein gewohntes Paktimport, nur dass das Schlüsselwort `static` eingefügt ist und erst dem klassennamen der Stern folgt, der in diesen Fall für alle statischen Eigenschaften steht.

Wir schreiben eine Hilfsklasse zum Arbeiten mit Strings, in der wir eine Methode zum umdrehen eines Strings vorsehen:

StringUtil.java

```
package name.panitz.staticImport;
public class StringUtil {
  static public String reverse(String arg) {
    StringBuffer result = new StringBuffer();
    for (char c:arg.toCharArray()) result.insert(0,c);
    return result.toString();
  }
}
```

Die Methode `reverse` wollen wir in einer anderen Klasse benutzen. Importieren wir die statischen Eigenschaften von `StringUtil`, so können wir auf die Qualifizierung des

Namens der Methode `reverse` verzichten:

UseStringUtil.java

```
package name.panitz.staticImport;
import static name.panitz.staticImport.StringUtil.*;
public class UseStringUtil {
  static public void main(String [] args) {
    for (String arg:args)
     System.out.println(reverse(arg));
  }
}
```

Die Ausgabe dieses programms:

```
> java -classpath classes/ name.panitz.staticImport.UseStringUtil hallo welt
ollah
tlew
>
```

4.2.8 Sichtbarkeitsattribute

ichtbarkeiten[8] erlauben es, zu kontrollieren, wer auf Klassen und ihre Eigenschaften zugreifen kann. Das *wer* bezieht sich hierbei auf andere Klassen und Pakete.

Sichtbarkeitsattribute für Klassen

Für Klassen gibt es zwei Möglichkeiten der Sichtbarkeit. Entweder darf von überall aus eine Klasse benutzt werden oder nur von Klassen im gleichen Paket. Syntaktisch wird dieses dadurch ausgedrückt, dass der Klassendefinition entweder das Schlüsselwort `public` vorangestellt ist oder aber kein solches Attribut voransteht:

MyPublicClass.java

```
package name.panitz.p1;
public class MyPublicClass {
}
```

MyNonPublicClass.java

```
package name.panitz.p1;
class MyNonPublicClass {
}
```

[8] Man findet in der Literatur auch den Ausdruck *Erreichbarkeiten*.

In einem anderen Paket dürfen wir nur die als öffentlich deklarierte Klasse benutzen. Folgende Klasse übersetzt fehlerfrei:

UsePublic.java

```
package name.panitz.p2;

import name.panitz.p1.*;

class UsePublic {
  public static void main(String [] args){
    System.out.println(new MyPublicClass());
  }
}
```

Der Versuch, eine nicht öffentliche Klasse aus einem anderen Paket heraus zu benutzen, gibt hingegen einen Übersetzungsfehler:

Fehlerhafter Code!

```
package name.panitz.p2;

import name.panitz.p1.*;

class UseNonPublic {
  public static void main(String [] args){
    System.out.println(new MyNonPublicClass());
  }
}
```

Java gibt bei der Übersetzung eine entsprechende gut verständliche Fehlermeldung:

```
sep@swe10:~> javac -d . UseNonPublic.java
UseNonPublic.java:7: name.panitz.p1.MyNonPublicClass is not
public in name.panitz.pantitz.p1;
cannot be accessed from outside package
        System.out.println(new MyNonPublicClass());
                                ^
UseNonPublic.java:7: MyNonPublicClass() is not
public in name.panitz.p1.MyNonPublicClass;
cannot be accessed from outside package
        System.out.println(new MyNonPublicClass());
                           ^
2 errors
sep@swe10:~>
```

Damit stellt Java eine Technik zur Verfügung, die es erlaubt, bestimmte Klassen eines Softwarepaketes als rein interne Klassen zu schreiben, die von außerhalb des Pakets nicht benutzt werden können.

Sichtbarkeitsattribute für Eigenschaften

Java stellt in Punkto Sichtbarkeiten eine noch feinere Granularität zur Verfügung. Es können nicht nur ganze Klassen als nicht-öffentlich deklariert, sondern für einzelne Eigenschaften von Klassen unterschiedliche Sichtbarkeiten deklariert werden.

Für Eigenschaften gibt es vier verschiedene Sichtbarkeiten:
`public`, `protected`, kein Attribut, `private`

Sichbarkeiten hängen zum einem von den Paketen ab, in denen sich die Klassen befinden, darüberhinaus unterscheiden sich Sichtbarkeiten auch darin, ob Klassen Unterklassen voneinander sind. Folgende Tabelle gibt eine Übersicht über die vier verschiedenen Sichtbarkeiten:

Attribut	Sichtbarkeit
`public`	Die Eigenschaft darf von jeder Klasse aus benutzt werden.
`protected`	Die Eigenschaft darf für jede Unterklasse und jede Klasse im gleichen Paket benutzt werden.
kein Attribut	Die Eigenschaft darf nur von Klassen im gleichen Paket benutzt werden.
`private`	Die Eigenschaft darf nur von der Klasse, in der sie definiert ist, benutzt werden.

Damit kann in einer Klasse auf Eigenschaften mit jeder dieser vier Sichtbarkeiten zugegriffen werden. Wir können die Fälle einmal systematisch durchprobieren. In einer öffentlichen Klasse eines Pakets `p1` definieren wir hierzu vier Felder mit den vier unterschiedlichen Sichtbarkeiten:

VisibilityOfFeatures.java

```
package name.panitz.p1;

public class VisibilityOfFeatures{
  private   String s1 = "private";
            String s2 = "package";
  protected String s3 = "protected";
  public    String s4 = "private";

  public static void main(String [] args){
    VisibilityOfFeatures v = new VisibilityOfFeatures();
    System.out.println(v.s1);
    System.out.println(v.s2);
    System.out.println(v.s3);
    System.out.println(v.s4);
  }
}
```

In der Klasse selbst können wir auf alle vier Felder zugreifen.

In einer anderen Klasse, die im gleichen Paket ist, können private Eigenschaften nicht mehr benutzt werden:

PrivateTest.java

```
package name.panitz.p1;

public class PrivateTest
{

  public static void main(String [] args){
    VisibilityOfFeatures v = new VisibilityOfFeatures();
    //s1 is private and cannot be accessed;
    //we are in a  different class.
    //System.out.println(v.s1);
    System.out.println(v.s2);
    System.out.println(v.s3);
    System.out.println(v.s4);
  }
}
```

Von einer Unterklasse können unabhängig von ihrem Paket die *geschützten* Eigenschaften benutzt werden. Ist die Unterklasse in einem anderen Paket, können Eigenschaften mit der Sichtbarkeit `package` nicht mehr benutzt werden:

PackageTest.java

```
package name.panitz.p2;
import name.panitz.p1.VisibilityOfFeatures;

public class  PackageTest extends VisibilityOfFeatures{

  public static void main(String [] args){
    PackageTest v = new PackageTest();
    //s1 is private and cannot be accessed
    // System.out.println(v.s1);

    //s2 is package visible and cannot be accessed;
    //we are in a different package.
    //System.out.println(v.s2);

    System.out.println(v.s3);
    System.out.println(v.s4);
  }
}
```

Von einer Klasse, die weder im gleichen Paket noch eine Unterklasse ist, können nur noch öffentliche Eigenschaften benutzt werden:

ProtectedTest.java

```
package name.panitz.p2;
import name.panitz.p1.VisibilityOfFeatures;

public class ProtectedTest  {

  public static void main(String [] args){
    VisibilityOfFeatures v = new VisibilityOfFeatures();
    //s1 is private and cannot be accessed
    // System.out.println(v.s1);

    //s2 is package visible and cannot be accessed. We are
    //in a different package
    //System.out.println(v.s2);

    //s2 is protected and cannot be accessed.
    //We are not a subclass
    //System.out.println(v.s3);

    System.out.println(v.s4);
  }
}
```

Java wird in seinem Sichtbarkeitskonzept oft kritisiert, und das von zwei Seiten. Einerseits ist es mit den vier Sichtbarkeiten schon relativ unübersichtlich; die verschiedenen Konzepte der Vererbung und der Pakete spielen bei Sichtbarkeiten eine Rolle. Andererseits ist es nicht vollständig genug und kann verschiedene denkbare Sichtbarkeiten nicht ausdrücken.

In der Praxis fällt die Entscheidung zwischen privaten und öffentlichen Eigenschaften leicht. Geschützte Eigenschaften sind hingegen selten. Das Gros der Eigenschaften hat die Standardsichtbarkeit der Paketsichtbarkeit.

4.3 Schnittstellen (Interfaces) und abstrakte Klassen

Wir haben schon Situationen kennengelernt, in denen wir eine Klasse geschrieben haben, von der nie ein Objekt konstruiert werden sollte, sondern für die wir nur Unterklassen definiert und instanziiert haben. Die Methoden in diesen Klassen hatten eine möglichst einfache Implementierung; sie sollten ja nie benutzt werden, sondern die überschreibenden Methoden in den Unterklassen. Ein Beispiel für eine solche Klassen war die Klasse `ButtonLogic`, mit der die Funktionalität eines GUIs definiert wurde.

Java bietet ein weiteres Konzept an, mit dem Methoden ohne eigentliche Implementierung deklariert werden können, die Schnittstellen.

4.3.1 Schnittstellen

Schnittstellendeklaration

Eine Schnittstelle sieht einer Klasse sehr ähnlich. Die syntaktischen Unterschiede sind:

- statt des Schlüsselworts `class` steht das Schlüsselwort `interface`.
- die Methoden haben keine Rümpfe, sondern nur eine Signatur.

So läßt sich für unsere Klasse `ButtonLogic` eine entsprechende Schnittstelle schreiben:

DialogueLogic.java

```
package name.panitz.dialoguegui;

public interface DialogueLogic {
  public String getDescription();
  public String eval(String input);
}
```

Schnittstellen sind ebenso wie Klassen mit dem Javaübersetzer zu übersetzen. Für Schnittstellen werden auch Klassendateien mit der Endung `.class` erzeugt.

Im Gegensatz zu Klassen haben Schnittstellen keinen Konstruktor. Das bedeutet insbesondere, dass mit einer Schnittstelle kein Objekt erzeugt werden kann. Was hätte ein solches Objekt auch für ein Verhalten? Die Methoden haben ja gar keinen Code, den sie ausführen könnten. Eine Schnittstelle ist vielmehr ein Versprechen, dass Objekte Methoden mit den in der Schnittstelle definierten Signaturen enthalten. Objekte können aber immer nur über Klassen erzeugt werden.

Implementierung von Schnittstellen

Objekte, die die Funktionalität einer Schnittstelle enthalten, können nur mit Klassen erzeugt werden, die diese Schnittstelle implementieren. Hierzu gibt es zusätzlich zur `extends`-Klausel in Klassen auch noch die Möglichkeit, eine `implements`-Klausel anzugeben.

Eine mögliche Implementierung der obigen Schnittstelle ist:

ToUpperCase.java

```
package name.panitz.dialoguegui;

public class ToUpperCase implements DialogueLogic{
  protected String result;

  public String getDescription(){
    return "convert into upper cases";
  }
  public String eval(String input){
    result = input.toUpperCase();
    return result;
  }
}
```

Die Klausel `implements DialogueLogic` verspricht, dass in dieser Klasse für alle Methoden aus der Schnittstelle eine Implementierung existiert. In unserem Beispiel waren zwei Methoden zu implementieren, die Methode `eval` und `getDescription()`.

Im Gegensatz zur `extends`-Klausel von Klassen können in einer `implements`-Klausel auch mehrere Schnittstellen angegeben werden, die implementiert werden.

Definieren wir zum Beispiel ein zweite Schnittstelle:

ToHTMLString.java

```
package name.panitz.html;

public interface ToHTMLString {
  public String toHTMLString();
}
```

Diese Schnittstelle verlangt, dass implementierende Klassen eine Methode haben, die für das Objekt eine Darstellung als HTML erzeugen können.

Jetzt können wir eine Klasse schreiben, die die beiden Schnittstellen implementiert.

ToUpper.java

```
package name.panitz.dialoguegui;

import name.panitz.html.*;

public class ToUpper extends    ToUpperCase
                     implements ToHTMLString, DialogueLogic {

  public String toHTMLString(){
     return   "<html><head><title>"+getDescription()
            + "</title></head>"
            + "<body><b>Small Gui application</b>"
            + " for convertion of "
            + " a <b>String</b> into <em>upper</em>"
            + " case letters.<br></br>"
            + "The result of your query was: <p>"
            + "<span style=\"font-family: monospace;\">"
            + result
            + "</span></p></body></html>";
  }
}
```

Schnittstellen können auch einander erweitern. Dieses geschieht dadurch, dass Schnittstellen auch eine `extends`-Klausel haben.

Wir können also auch eine Schnittstelle definieren, die die beiden obigen Schnittstellen zusammenfaßt:

DialogueLogics.java

```
package name.panitz.dialoguegui;

import name.panitz.html.*;

public interface DialogueLogics extends ToHTMLString, DialogueLogic {}
```

Ebenso können wir jetzt eine Klasse ableiten, die diese Schnittstelle implementiert:

UpperConversion.java

```
package name.panitz.dialoguegui;

public class UpperConversion extends ToUpper
                      implements DialogueLogics{}
```

Benutzung von Schnittstellen

Schnittstellen sind genauso Typen wie Klassen. Wir kennen bisher also die folgenden Arten von Typen:

- primitive Typen
- Klassen
- Schnittstellen
- Reihungen

Parameter können vom Typ einer Schnittstellen sein, ebenso wie Felder oder Rückgabetypen von Methoden. Die Zuweisungkompatibilität nutzt nicht nur die Unterklassenbeziehung, sondern auch die Implementierungsbeziehung. Ein Objekt der Klasse `C` darf einem Feld des Typs der Schnittstelle `I` zugewiesen werden, wenn `C` die Schnittstelle `I` implementiert.

Im Folgenden eine kleine Gui-Anwendung, die wir im einzelnen noch nicht verstehen müssen. Man beachte, dass der Typ `DialogueLogics` an mehreren Stellen benutzt wird wie ein ganz normaler Klassentyp. Nur einen Konstruktoraufruf mit `new` können wir für diesen Typ nicht machen.

HtmlDialogue.java

```
package name.panitz.dialoguegui;
import java.awt.*;
import javax.swing.*;
import javax.swing.text.html.*;

public class HtmlDialogue extends JFrame{
  final JTextField inputField = new JTextField(20) ;
  final JTextPane outputField = new JTextPane();

  public HtmlDialogue(DialogueLogics l){
    outputField.setEditorKit(new HTMLEditorKit());
    JButton button=new JButton(l.getDescription());
    button.addActionListener(ev -> {
          l.eval(inputField.getText().trim());
          outputField.setText(l.toHTMLString());
          pack();
        } );
    JPanel p = new JPanel(new BorderLayout());
    p.add(inputField,BorderLayout.NORTH);
    p.add(button,BorderLayout.CENTER);
    p.add(outputField,BorderLayout.SOUTH);
    add(p);    pack();   setVisible(true);
  }
}
```

Schließlich können wir ein Objekt der Klasse `UpperConversion`, die die Schnittstelle `DialogueLogics` implementiert, konstruieren und der Gui-Anwendung übergeben:

HtmlDialogueTest.java

```
package name.panitz.dialoguegui;
public class HtmlDialogueTest {
  public static void main(String [] args){
    new HtmlDialogue(new UpperConversion());
  }
}
```

Die Anwendung in voller Aktion kann in Abbildung 4.1 bewundert werden.

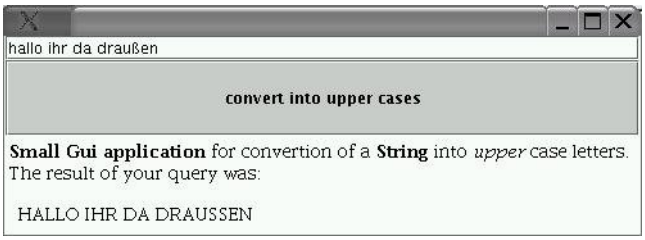

Abbildung 4.1: Ein Gui-Dialog mit Html-Ausgabe.

Default-Methoden

Schnittstellen können auch konkrete Methoden enthalten. Diese sind aber zu markieren. mit dem Attribut `default`. Man nennt sie default- oder Standardmethoden der Schnittstelle. Wenn eine Klasse geschrieben wird, die die Schnittstelle implementietr, dann bekommt sie automatisch die Standardmethoden dieser Schnittstelle hinzugefügt. Innerhalb einer Standardmethode können bereits abstrakte Methoden der Schnittstelle aufgerufen werden.

Als Beispiel schreiben wir eine Schnittstelle mit einer abstrakten und einer Standardmethode:

```
shell> interface Gleich{
   ...>   boolean istGleichZu(Object that);
   ...>   default boolean istNichtGleichZu(Object that){
   ...>     return !istGleichZu(that);
   ...>   }
   ...> }
|  created interface Gleich
```

Die abstrakte Methode heißt `istGleichZu`. Unter der Annahme, dass diese durch eine implementierende Klasse umgesetzt wird, können wir schon einmal als Standardmethode

implementieren, wie die Methode `istNichtGleichZu` rechnen soll.

Eine Klasse, die diese Schnittstelle implementiert, muss nur die Methode `istGleichZu` umsetzen.

```
jshell> class Apfel implements Gleich{
   ...>   @Override public boolean istGleichZu(Object that){
   ...>     return that instanceof Apfel;
   ...>   }
   ...> }
|  created class Apfel
```

Objekte dieser Klasse bekommen automatisch auch die Methode `istNichtGleichZu` geschenkt.

```
jshell> new Apfel().istNichtGleichZu("Birnenstring")
$21 ==> true

jshell> new Apfel().istNichtGleichZu(new Apfel())
$22 ==> false
```

Statische Methoden

Schnittstellen können auch statische Methoden enthalten. Diese sind konkret und zählen nicht zu den abstrakten Methoden der Schnittstelle.

```
jshell> interface SayHello{
   ...>   static String hello(){
   ...>     return "Hello!";
   ...>   }
   ...> }
|  created interface SayHello
```

In einer Klasse, die die Schnittstelle implementiert, bekommt man aber nicht automatisch die statische Methode hinzu. Will man die Methode `hello` verwenden, so muss man explizit angeben, dass sie aus der Schnittstelle `SayHello` ist.

```
jshell> class A implements SayHello{
   ...>   static void sayIt(){
   ...>     System.out.println(SayHello.hello());
   ...>   }
   ...>   static String hello(){return "Servus!";}
   ...> }
|  replaced class A

jshell> A.sayIt()
Hello!
```

Man kann in der implementierenden Klasse auch eine statische Methode mit gleichem Namen schreiben. Diese kann dann auch ohne Angabe der Klasse, in der die Methode sich befindet, aufgerufen werden.

```
jshell> class A implements SayHello{
   ...>   static void sayIt(){
   ...>     System.out.println(hello());
   ...>   }
   ...>   static String hello(){return "Servus!";}
   ...> }
|  modified class A

jshell> A.sayIt()
Servus!
```

Statische Methoden werden aber nicht in irgendeiner Weise vererbt. In der Klasse `A` existiert nicht eine geerbte statische Methode `hello()` aus der Schnittstelle `SayHello`. Es kommt hier zu einen Kompilatorfehler.

```
jshell> class A implements SayHello{
   ...>   void sayIt(){
   ...>     System.out.println(A.hello());
   ...>   }
   ...>
   ...> }
|  modified class A, however, it cannot be instantiated or its methods invok
ed until method hello() is declared
```

Funktionale Schnittstellen

Es gibt in Java eine besondere Art von ausgezeichneten Schnittstellen, die sogenannten funktionalen Schnittstellen. Eine Schnittstelle ist eine funktionale Schnittstelle, wenn sie genau eine abstrakte Methode enthält. Wenn eine funktionale Schnittstelle durvch eine Klasse zu implementieren ist, muss genau eine Methode implementiert werden.

Man kann eine funktionale Schnittstelle als eine solche markieren, indem man die Annotation `@FunctionalInterface` hinzufügt. Dann überprüft der Compiler, ob die Schnittstelle auch tatsächlich nur eine abstrakte Methode hat. Die Annotation hat aber keine weitere Bedeutung zur Laufzeit.

Beispielsweise hier eine funktionale Schnittstelle. Die hat genau eine abstrakte Methode, die Methode `wandelStringUm`.

```
jshell> @FunctionalInterface interface StringHer{
   ...>   String wandelStringUm(String s);
   ...> }
|  created interface StringHer
```

Wenn ein Objekt erzeugt werden soll, das die Schnittstelle `StringHer` implementiert, ist technisch erst eine Klasse zu schreiben, in der die abstrakte Methode konkret implementiert wird, und dann für diese Klasse ein Objekt zu erzeugen. Das ist viel Aufwand dafür, dass nur eine Methode, die ein String-Argument und ein String-Ergebnis hat, umzusetzen ist.

Daher gibt es für diesen ganzen Prozess eine Kurzschreibweise, die sogenannte -Ausdrücke. Diese konzentrieren sich alleine darauf, genau eine Methode zu schreiben. Syntaktisch wird der Pfeil in Form `->` verwendet. Vor diesem Pfeil stehen die formalen Parameter der Methode, nach diesem Pfeil der Ausdruck, der das Ergebnis berechnet. In unseren Beispiel können wir jetzt mit wenig Aufwand ein Objekt des Typs `StringHer` erzeugen:

```
jshell> StringHer toUpper = (x) -> x.toUpperCase()
toUpper ==> $Lambda$142/0x0000000800c44840@5dd6264
```

Wir implementieren hier die Methode `wandelStringUm`, ohne diese namentlich zu nennen. Wir schreiben nicht einmal für den formalen Parameter `x` die Typisierung hin. Wir verwenden nicht einmal die `return`-Anweisung. Das Objekt kann nun ganz normal wie ein Objekt verwendet werden:

```
jshell> toUpper.wandelStringUm("burgstraße")
$25 ==> "BURGSTRASSE"
```

Wem das alles zu wenig Information im konkreten Quelltext ist, und wer lieber die Typen sehen will, kann die Methode etwas verboser schreiben, indem der Parameter typisiert wird ind nach dem Pfeilsymbol auch einen richtigen Methodenrumpf verwendet.

```
jshell> StringHer toUpper = (String x) -> {return x.toUpperCase();}
toUpper ==> $Lambda$142/0x0000000800c44840@5dd6264

jshell> toUpper.wandelStringUm("burgstraße")
```

Zusammenfassung: Eigenschaften von Schnittstellen

Es gibt einige semantische Einschränkungen, die über die syntaktischen Einschränkungen hinausgehen:

- Schnittstellen können nur Schnittstellen, nicht aber Klassen erweitern.
- Jede Methode einer Schnittstelle muss öffentlich sein, braucht also das Attribut `public`. Wenn dieses für eine Methode nicht deklariert ist, so wird Java dieses von selbst hinzufügen. Trotzdem müssen implementierende Klassen diese Methode dann als öffentlich deklarieren. Daher ist es besser, das Attribut `public` auch hinzuschreiben.
- Schnittstellen haben keine Konstruktoren.
- Jede Methode ist abstrakt, d.h. hat keinen Rumpf. Man kann dieses noch zusätzlich deutlich machen, indem man das Attribut `abstract` für die Methode mit angibt. Ausgenommen sind default-Methoden in der Schnittstelle. Diese sind nicht mehr abstrakt und haben einen Methodenrumpf.
- Felder einer Schnittstelle sind immer statisch, brauchen also das Attribut `static` und zusätzlich noch das Attribut `final`.

4.3.2 Abstrakte Klassen

Im vorangegangenen Abschnitt haben wir zusätzlich zu Klassen noch das Konzept der Schnittstellen kennengelernt. Klassen enthalten Methoden und Implementierungen für die Methoden. Jede Methode hat einen Rumpf. Schnittstellen enthalten nur Methodensignaturen. Keine Methode einer Schnittstelle außer der speziellen default-Methoden hat eine Implementierung. Es gibt keine Methodenrümpfe. Java kennt noch eine Mischform zwischen Klassen und Schnittstellen: abstrakte Klassen.

Definition abstrakter Klassen

Eine Klasse wird als abstrakt deklariert, indem dem Schlüssenwort `class` das Schlüsselwort `abstract` vorangestellt wird. Eine abstrakte Klasse kann nun Methoden mit und Methoden ohne Rumpf enthalten. Methoden, die in einer abstrakten Klassen keine Implementierung enthalten, sind mit dem Attribut `abstract` zu kennzeichnen. Für abstrakte Klassen gilt also, daß bestimmte Methoden bereits implementiert sind und andere Methoden in Unterklassen zu implementieren sind.

5 Graphische Benutzeroberflächen mit Swing

Die meisten Programme auf heutigen Rechnersystemen haben eine graphische Benutzeroberfläche (GUI)[1].

Java stellt Klassen zur Verfügung, mit denen graphische Objekte erzeugt und in ihrem Verhalten instrumentalisiert werden können. Es gibt mehrere Bibliotheken in Java, die Klassen bereit stellen, um GUIs zu implementieren. Ganz ursprünglich gab es in Java die AWT Bibliothek. Deren Klassen liegen im Paket `java.awt`. Doch schon nach wenigen Jahren wurde eine neue Bibliothek entwickelt, die sogenannte Swing GUI-Bibliothek. Deren Klassen befinden sich im Paket `javax.swing`. Mittlerweile soll langfristig Swing nicht mehr weiter entwickelt werden, sondern in Zukunft die Bibliothek Java FX der Standard zur GUI Programmierung in Java sein. Es gibt aber auch externe GUI-Bibliotheken. Hier ist insbesondere http://www.eclipse.org/swt/http://www.eclipse.org/swt/ zu nennen. Auch das google web toolkit, das zur Entwicklung von Webapplikationen von der Firma Google zur Verfügung gestellt wird, enthält eine GUI-Bibliothek. Dort wird die entwickelte Benutzerschnittstelle so übersetzt, dass sie in einem Browser läuft.

Wir werden uns in diesem Kapitel als Beispiel der Swing Bibliothek widmen.

Leider kommt man bei der Swing Programmierung nicht darum herum, auch Klassen aus der AWT Bibliothek zu verwenden. Die beiden Pakete überschneiden sich in der Funktionalität.

- `java.awt`: Dieses ist das ältere Paket zur GUI-Programmierung. Es enthält Klassen für viele graphische Objekte (z.B. eine Klasse `Button`) und Unterpakete, zur Programmierung der Funktionalität der einzelnen Komponenten.
- `javax.swing`: Dieses neuere Paket ist noch universeller und platformunabhängiger als das `java.awt`-Paket. Auch hier finden sich Klassen für unterschiedliche GUI-Komponenten. Sie entsprechen den Klassen aus dem Paket `java.awt`. Die Klassen haben oft den gleichen Klassennamen wie in `java.awt` jedoch mit einem `J` vorangestellt. So gib es z.B. eine Klasse `JButton`.

Man ist angehalten, sofern man sich für eine Implementierung seines GUIs mit den Paket `javax.swing` entschieden hat, nur die graphischen Komponenten aus diesem Paket zu benutzen; die Klassen leiten aber von Klassen des Pakets `java.awt` ab. Hinzu kommt,

[1] GUI ist die Abkürzung für *graphical user interface*. Entsprechend wäre GRABO eine Abkürzung für das deutsche *graphische Benutzeroberfläche*.

dass die Ereignisklassen, die die Funktionalität graphischer Objekte bestimmen, nur in `java.awt` existieren und nicht noch einmal für `javax.swing` extra umgesetzt wurden.

Sind Sie verwirrt? Sie werden es hoffentlich nicht mehr sein, nachdem Sie die Beispiele dieses Kapitels durchgespielt haben.

Zur Programmierung eines GUIs müssen drei fundamentale Fragestellungen gelöst werden:

- Welche graphischen Komponenten stehen zur Verfügung? Knöpfe, Textfelder, Baumdarstellugen, Tabellen....
- Wie können diese Komponenten angeordnet und gruppiert werden? Was gibt es für Layout-Möglichkeiten?
- Wie reagiert man auf Ereignisse, zum Beispiel auf einen Mausklick?

Javas Swing Bibliothek kennt für die Aufgaben:

- Komponenten wie `JButton`, `JTextField`, `JLabel`,...
- Layout Manager und Komponenten, um andere Komponenten zu gruppieren, z.B. `JPanel`
- Event Handler, in denen implementiert wird, wie auf ein Ereignis reagiert werden soll.

In den folgenden Abschnitten, werden wir an ausgewählten Beispielen Klassen für diese drei wichtigen Schritte der GUI-Programmierung kennenlernen.

5.1 Swings GUI-Komponenten

Javas `swing`-Paket kennt drei Arten von Komponenten.

- Top-Level Komponenten
- Zwischenkomponenten
- Atomare Komponenten

Leider spiegelt sich diese Unterscheidung nicht in der Ableitungshierarchie wider. Alle Komponenten leiten schließlich von der Klasse `java.awt.Component` ab. Es gibt keine Schnittmengen, die Beschreiben, dass bestimmte Komponenten atomar oder top-level sind.

Komponenten können Unterkomponenten enthalten; ein Fenster kann z.B. verschiedene Knöpfe und Textflächen als Unterkomponenten enthalten.

5.1.1 Top-Level Komponenten

Eine top-level Komponenten ist ein GUI-Objekt, das weitere graphische Objekte enthalten kann, selbst aber kein graphisches Objekt hat, in dem es enthalten ist. Somit sind

top-level Komponenten in der Regel Fenster, die weitere Komponenten als Fensterinhalt haben. Swing top-level Komponenten sind Fenster und Dialogfenster. Hierfür stellt Swing entsprechende Klassen zur Verfügung: `JFrame, JDialog` Eine weitere top-level Komponente steht für *Applets* zur Verfügung: `JApplet`.

Graphische Komponenten haben Konstruktoren, mit denen sie erzeugt werden. Für die Klasse `JFrame` existiert ein parameterloser Konstruktor.

Das minimalste GUI-Programm ist wahrscheinlich folgendes Programm, das ein Fensterobjekt erzeugt und dieses sichtbar macht.

JF.java

```
package name.panitz.oose.swing.example;
import javax.swing.*;

class JF {
  public static void main(String [] args){
    new JFrame().setVisible(true);
  }
}
```

Dieses Programm erzeugt ein leeres Fenster und gibt das auf dem Bildschirm aus.

Die Klasse `JFrame` hat einen zweiten Konstruktor, der noch ein Stringargument hat. Der übergebene String wird als Fenstertiteltext benutzt.

Ein Programm, kann auch mehrere Fensterobjekte erzeugen und sichtbar machen.

JF2.java

```
package name.panitz.oose.swing.example;
import javax.swing.*;

class JF2 {
  public static void main(String [] args){
    new JFrame("erster Rahmen").setVisible(true);
    new JFrame("zweiter Rahmen").setVisible(true);
  }
}
```

5.1.2 Zwischenkomponenten

Graphische Zwischenkomponenten haben primär die Aufgabe andere Komponenten als Unterkomponenten zu haben und diese in einer bestimmten Weise anzuordnen. Zwischenkomponenten haben oft keine eigene visuelle Ausprägung. Sie sind dann unsichtbare Komponenten, die als Behälter weiterer Komponenten dienen.

Die gebräuchlichste Zwischenkomponenten ist von der Klasse `JPanel`. Weitere Zwischenkomponenten sind `JScrollPane` und `JTabbedPane`. Diese haben auch eine eigene visuelle Ausprägung.

5.1.3 Atomare Komponenten

Die atomaren Komponenten sind schließlich Komponenten, die ein konkretes graphisches Objekt darstellen, das keine weiteren Unterkomponenten enthalten kann. Solche Komponenten sind z.B.
`JButton, JTextField, JTable` und `JComBox`.

Diese Komponenten lassen sich über ihre Konstruktoren instanziieren, um sie dann einer Zwischenkomponenten über deren Methode `add` als Unterkomponente hinzuzufügen. Sind alle gewünschten graphischen Objekte einer Zwischenkomponente hinzugefügt worden, so kann auf der zugehörigen top-level Komponenten die Methode `pack` aufgerufen wurden. Diese berechnet die notwendige Größe und das Layout des Fensters, welches schließlich sichtbar gemacht wird.

Das folgende Programm erzeugt ein Fenster mit einer Textfläche.

```
import javax.swing.*;

class JT {
  public static void main(String [] args){
    JFrame frame = new JFrame();
    JTextArea textArea = new JTextArea();
    textArea.setText("hallo da draußen");
    frame.add(textArea);
    frame.pack();
    frame.setVisible(true);
  }
}
```

In der Regel wird man die GUI-Komponente, die man schreibt nicht nacheinander in der Hauptmethode definieren, sondern es wird ein eigenes Objekt definiert, dass die GUI Komponente darstellt. Hierzu leitet man eine spezifische GUI-Klasse von der Klasse `JPanel` ab und fügt bereits im Konstruktor die entsprechenden Unterkomponenten hinzu.

Die folgende Klasse definiert eine Komponente, die einen Knopf und eine Textfläche enthält. In der Hauptmethode wird das Objekt instantiiert:

JTB.java

```
package name.panitz.simpleGui;
import javax.swing.*;

class JTB extends JPanel {
  JTextArea textArea = new JTextArea();
  JButton button = new JButton("ein knopf");

  public JTB(){
    textArea.setText("hallo da draußen");
    add(textArea);
    add(button);
  }

  public void showInFrame(){
    JFrame f = new JFrame();
    f.add(this);
    f.pack();
    f.setVisible(true);
  }

 public static void main(String [] args){
    new JTB().showInFrame();
  }
}
```

5.2 Gruppierungen

Bisher haben wir Unterkomponenten weitere Komponenten mit der Methode `add` hinzugefügt, ohne uns Gedanken über die Plazierung der Komponenten zu machen. Wir haben einfach auf das Standardverhalten zur Plazierung von Komponenten vertraut. Ob die Komponenten schließlich nebeneinander, übereinander oder irgendwie anders Gruppiert im Fenster erschienen, haben wir nicht spezifiziert.

Um das Layout von graphischen Komponenten zu steuern, steht das Konzept der sogenannten *Layout-Manager* zur Verfügung. Ein Layout-Manager ist ein Objekt, das einer Komponente hinzugefügt wird. Der Layout-Manager steuert dann, in welcher Weise die Unterkomponenten gruppiert werden.

`LayoutManager` ist eine Schnittstelle. Es gibt mehrere Implementierungen dieser Schnittstelle. Wir werden in den nächsten Abschnitten drei davon kennenlernen. Es steht einem natürlich frei, eigene Layout-Manager durch Implementierung dieser Schnittstelle zu schreiben. Es wird aber davon abgeraten, weil dieses notorisch schwierig ist und die in Java bereits vorhandenen Layout-Manager bereits sehr mächtig und ausdrucksstark sind.

Zum Hinzufügen eines Layout-Manager gibt es die Methode `setLayout`.

5.2.1 Flow Layout

Der vielleicht einfachste Layout-Manager nennt sich `FlowLayout`. Hier werden die Unterkomponenten einfach der Reihe nach in einer Zeile angeordnet. Erst wenn das Fenster zu schmal hierzu ist, werden weitere Komponenten in eine neue Zeile gruppiert.

Die folgende Klasse definiert ein Fenster, dessen Layout über ein Objekt der Klasse `FlowLayout` gesteuert wird. Dem Fenster werden fünf Knöpfe hinzugefügt:

FlowLayoutTest.java

```
package name.panitz.gui.layoutTest;

import java.awt.*;
import javax.swing.*;

class FlowLayoutTest extends JFrame {

  public FlowLayoutTest(){
    JPanel pane = new JPanel();
    pane.setLayout(new FlowLayout());
    pane.add(new JButton("eins"));
    pane.add(new JButton("zwei"));
    pane.add(new JButton("drei (ein langer Knopf)"));
    pane.add(new JButton("vier"));
    pane.add(new JButton("fuenf"));
    add(pane);
    pack();
    setVisible(true);
  }

  public static void main(String [] args){new FlowLayoutTest();}
}
```

Das Fenster hat die optische Ausprägung aus Abbildung ??.

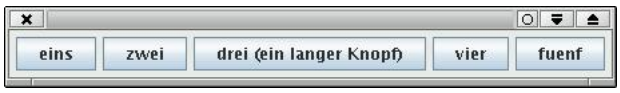

Abbildung 5.1: Anordnung durch das Flow Layout.

Verändert man mit der Maus die Fenstergröße, macht es z.B. schmal und hoch, so werden die Knöpfe nicht mehr nebeneinander sonder übereinander angeordnet.

5.2.2 Border Layout

Die Klasse `BorderLayout` definiert einen Layout Manager, der fünf feste Positionen kennt: eine Zentralposition, und jeweils links/rechts und oberhalb/unterhalb der Zentralposition eine Position für Unterkomponenten. Die Methode `add` kann in diesem Layout auch noch mit einem zweitem Argument aufgerufen werden, das eine dieser fünf Positionen angibt. Hierzu bedient man sich der konstanten Felder der Klasse `BorderLayout`.

In dieser Klasse wird die Klasse `BorderLayout` zur Steuerung des Layoout benutzt. Die fünf Knöpfe werden an jeweils eine der fünf Positionen hinzugefügt:

BorderLayoutTest.java

```
package name.panitz.gui.layoutTest;

import java.awt.*;
import javax.swing.*;

class BorderLayoutTest extends JFrame {

  public BorderLayoutTest(){
    JPanel pane = new JPanel();
    pane.setLayout(new BorderLayout());
    pane.add(new JButton("eins"),BorderLayout.NORTH);
    pane.add(new JButton("zwei"),BorderLayout.SOUTH);
    pane.add(new JButton("drei (ein langer Knopf)")
                                  ,BorderLayout.CENTER);
    pane.add(new JButton("vier"),BorderLayout.WEST);
    pane.add(new JButton("fuenf"),BorderLayout.EAST);
    add(pane);
    pack();
    setVisible(true);
  }

  public static void main(String [] args){new BorderLayoutTest();}
}
```

Die Klasse erzeugt das Fenster aus Abbildung ??.

Abbildung 5.2: Anordnung über das Border Layout

Das Layout ändert sich nicht, wenn man mit der Maus die Größe und das Format des

Fensters verändert.

5.2.3 Grid Layout

Die Klasse `GridLayout` ordnet die Unterkomponenten tabellarisch an. Jede Komponente wird dabei gleich groß ausgerichtet. Die Größe richtet sich also nach dem größten Element.

Folgende Klasse benutzt ein Grid-Layout mit zwei Zeilen zu je drei Spalten.

GridLayoutTest.java

```
package name.panitz.gui.layoutTest;

import java.awt.*;
import javax.swing.*;

class GridLayoutTest extends JFrame {
  public GridLayoutTest(){
    JPanel pane = new JPanel();
    pane.setLayout(new GridLayout(2,3));
    pane.add(new JButton("eins"));
    pane.add(new JButton("zwei"));
    pane.add(new JButton("drei (ein langer Knopf)"));
    pane.add(new JButton("vier"));
    pane.add(new JButton("fünf"));
    add(pane);
    pack();
    setVisible(true);
  }

  public static void main(String [] args){new GridLayoutTest();}
}
```

Folgendes Fensters wird durch dieses Programm geöffnet.

Abbildung 5.3: Anordnung im Grid Layout

Auch hier ändert sich das Layout nicht, wenn man mit der Maus die Größe und das Format des Fensters verändert.

5.3 Eigene GUI-Komponenten

Bisher graphische Komponenten unter Verwendung der fertigen GUI-Komponente aus der Swing-Bibliothek zusammengesetzt. Oft will man graphische Komponenten schreiben, für die es keine fertige GUI-Komponente in der Swing-Bibliothek gibt. Wir müssen wir eine entsprechende Komponente selbst schreiben.

Um eine eigene GUI-Komponente zu schreiben, schreibt man eine Klasse, die von der GUI-Klasse ableitet. Dieses haben wir bereits in den letzten Beispielen getan, indem wir von der Klasse `JFrame` abgelitten haben. Die dort geschriebenen Unterklassen der Klasse `JFrame` zeichneten sich dadurch aus, daß sie eine Menge von graphischen Objekten (Knöpfe, Textfelder...) in einer Komponente zusammengefasst haben. In diesem Abschnitt werden wir eine neue Komponente definieren, die keine der bestehenden fertigen Komponenten benutzt, sondern selbst alles zeichnet, was zu ihrer Darstellung notwendig ist.

Hierzu betrachten wir eine der entscheidenen Methoden der Klasse `JComponent`, die Methode `paintComponent`. In dieser Methode wird festgelegt, was zu zeichnen ist, wenn die graphische Komponente darzustellen ist. Die Methode `paintComponent` hat folgende Signatur:

```
public void paintComponent(java.awt.Graphics g)
```

Java ruft diese Methode immer auf, wenn die graphische Komponente aus irgendeinen Grund zu zeichnen ist. Dabei bekommt die Methode das Objekt übergeben, auf dem gezeichnet wird. Dieses Objekt ist vom Typ `java.awt.Graphics`. Es stellt ein zweidimensionales Koordinatensystem dar, in dem zweidimensionale Graphiken gezeichnet werden können. Der Nullpunkt dieses Koordinatensystems ist oben links und nicht unten links, wie wir es vielleicht aus der Mathematik erwartet hätten.

In der Klasse `Graphics` sind eine Reihe von Methoden definiert, die es erlauben graphische Objekte zu zeichnen. Es gibt Methoden zum Zeichnen von Geraden, Vierecken, Ovalen, beliebigen Polygonzügen, Texten etc.

Wollen wir eine eigene graphische Komponente definieren, so können wir die Methode `paintComponent` überschreiben und auf dem übergebenen Objekt des Typs `Graphics` entsprechende Methoden zum Zeichnen aufrufen. Um eine eigene graphische Komponente zu definieren, wird empfohlen die Klasse `JPanel` zu erweitern und in ihr die Methode `paintComponent` zu überschreiben.

Folgende Klasse definiert eine neue graphische Komponente, die zwei Linien, einen Text, ein Rechteck, ein Oval und ein gefülltes Kreissegment enthält.

SimpleGraphics.java

```
package name.panitz.gui.graphicsTest;

import javax.swing.JPanel;
import javax.swing.JFrame;
import java.awt.Graphics;

class SimpleGraphics extends JPanel{
  public void paintComponent(Graphics g){
    g.drawLine(0,0,100,200);
    g.drawLine(0,50,100,50);
    g.drawString("hallo",10,20);
    g.drawRect(10, 10, 60,130);
    g.drawOval( 50,  100, 30, 80);
    g.fillArc(-20, 150, 80, 80, 0, 50);
  }
}
```

Diese Komponente können wir wie jede andere Komponente auch einem Fenster hinzufügen, so daß sie auf dem Bildschirm angezeigt werden kann.

UseSimpleGraphics.java

```
package name.panitz.gui.graphicsTest;

import javax.swing.JFrame;

class UseSimpleGraphics {
  public static void main(String [] args){
    JFrame frame = new JFrame();
    frame.getContentPane().add(new SimpleGraphics());

    frame.pack();
    frame.setVisible(true);
  }
}
```

Ärgerlich in unserem letzten Beispiel war, daß Java zunächst ein zu kleines Fenster für unsere Komponente geöffnet hat, und wir dieses Fenster mit Maus erst größer ziehen mußten. Die Klasse `JComponent` enthält Methoden, in denen die Objekte angeben können, welches ihre bevorzugte Größe bei ihrere Darstellung ist. Wenn wir diese Methode überschreiben, so daß sie eine Dimension zurückgibt, in der das ganze zu zeichnende Bild passt, so wird von Java auch ein entsprechend großes Fenster geöffnet. Wir fügen der Klasse `SimpleGraphics` folgende zusätzliche Methode hinzu.

```
  public java.awt.Dimension getPreferredSize() {
    return new java.awt.Dimension(100,200);
  }
```

Jetzt öffnet Java ein Fenster, in dem das ganze Bild dargestellt werden kann.

5.3.1 Fraktale

Um noch ein wenig mit Farben zu spielen, zeichnen wir in diesem Abschnitt die berühmten Apfelmännchen. Apfelmännchen werden definiert über eine Funktion auf komplexen Zahlen. Die aus der Mathematik bekannten komplexen Zahlen sind Zahlen mit zwei reellen Zahlen als Bestandteil, den sogenannten Imaginärteil und den sogenannten Realteil. Wir schreiben zunächst eine rudimentäre Klasse zur Darstellung von komplexen Zahlen:

Complex.java

```
package name.panitz.crempel.tool.apfel;

public class Complex{
```

Diese Klasse braucht zwei Felder um Real- und Imaginärteil zu speichern:

Complex.java

```
  public double re;
  public double im;
```

Ein naheliegender Konstruktor für komplexe Zahlen füllt diese beiden Felder.

Complex.java

```
 public Complex(double re,double im){
  this.re=re;this.im=im;
}
```

Im Mathematikbuch schauen wir nach, wie Addition und Multiplikation für komplexe Zahlen definiert sind, und schreiben entsprechende Methoden:

Complex.java

```
  public Complex add(Complex other){
    return new Complex(re+other.re,im+other.im);
  }

  public Complex mult(Complex other){
    return new Complex
      (re*other.re-im*other.im,re*other.im+im*other.re);
  }
```

Zusätzlich finden wir in Mathematik noch die Definition der Norm einer komplexen Zahl und setzen auch diese Definition in eine Methode um. Zum Quadrat des Realteils wird das Quadrat des Imaginärteils addiert.

Complex.java

```
  public double norm(){return re*re+im*im;}
}
```

Soweit komplexe Zahlen, wie wir sie für Apfelmännchen brauchen.

Grundlage zum Zeichnen von Apfelmännchen ist folgende Iterationsgleichung auf komplexen Zahlen: $z_{n+1} = z_n^2 + c$. Wobei z_0 die komplexe Zahl $0 + 0i$ mit dem Real- und Imaginärteil 0 ist.

Zum Zeichnen der Apfelmännchen wird ein Koordinatensystem so interpretiert, daß die Achsen jeweils Real- und Imaginärteil von komplexen Zahlen darstellen. Jeder Punkt in diesem Koordinatensystem steht jetzt für die Konstante c in obiger Gleichung. Nun wir geprüft ob und für welches n die Norm von z_n größer eines bestimmten Schwellwertes ist. Je nach der Größe von $n$wird der Punkt im Koordinatensystem mit einer anderen Farbe eingefärbt.

Mit diesem Wissen können wir nun versuchen die Apfelmännchen zu zeichnen. Wir müssen nur geeignete Werte für die einzelnen Parameter finden. Wir schreiben eine eigene Klasse für das graphische Objekt, in dem ein Apfelmännchen gezeichnet wird. Wir deklarieren die Imports der benötigten Klassen:

Apfelmaennchen.java

```
package name.panitz.crempel.tool.apfel;

import java.awt.Graphics;
import java.awt.Color;
import java.awt.Dimension;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Apfelmaennchen extends JPanel {
```

Als erstes deklarieren wir Konstanten für die Größe des Apfelmännchens.

Apfelmaennchen.java

```
final int width = 480;
final int height = 430;
```

Eine weitere wichtige Konstante ist der Faktor, der angibt, welcher reellen Zahl ein Pixel entspricht:

Apfelmaennchen.java

```
double zelle=0.00625;
```

Eine weitere Konstanten legt die Farbe fest, mit der die Punkte, die nicht über einen bestimmten Schwellwert konvergieren, eingefärbt werden sollen:

Apfelmaennchen.java

```
final Color colAppleman = new Color(0,129,190);
```

Weitere Konstanten legen fest welche komplexe Zahl der Nullpunkt unseres `Graphics`-Objekts darstellt.

Apfelmaennchen.java

```
double startX = -2;
double startY = -1.35;
```

Weitere Konstanten sind der Schwellwert und die maximale Rekursionstiefe n, für die wir jeweils z_n berechnen:

Apfelmaennchen.java

```
final int recDepth = 50;
final int schwellwert = 4;
```

Die wichtigste Methode berechnet die Werte für die Gleichung $z_{n+1} = z_n^2 + c$. Der Eingabeparameter ist die komplexe Zahl c. Das Ergebnis dieser Methode ist das n, für das z_n größer als der Schwellwert ist:

Apfelmaennchen.java

```
//C-Werte checken nach zn+1 = zn*zn + c,
public int checkC(Complex c) {
  Complex zn = new Complex(0,0);

  for (int n=0;n<recDepth;n=n+1) {
    final Complex znp1 = zn.mult(zn).add(c);
    if (znp1.norm() > schwellwert) return n;
    zn=znp1;
  }
  return recDepth;
}
```

Jetzt gehen wir zum Zeichnen jedes Pixel unseres `Graphics`-Objekts durch, berechnen welche komplexe Zahl an dieser Stelle steht und benutzen dann die Methode *checkC*, um zu berechnen ob und nach wieviel Iterationen die Norm von z_n größer als der Schwellwert wird. Abhängig von dieser Zahl, färben wir den Punkt mit einer Farbe ein.

Apfelmaennchen.java

```
public void paint(Graphics g) {
  for (int y=0;y<height;y=y+1) {
    for (int x=0;x<width;x=x+1) {

      final Complex current
        =new Complex(startX+x*zelle,startY+y*zelle);

      final int iterationenC = checkC(current);

      paintColorPoint(x,y,iterationenC,g);
    }
  }
}
```

Zur Auswahl der Farbe benutzen wir folgende kleine Methode, die Abhängig von ihrem Parameter `it` an der Stelle `(x,y)` einen Punkt in einer bestimmten Farbe zeichnet.

Apfelmaennchen.java

```
  private void paintColorPoint
                (int x,int y,int it,Graphics g){
    final Color col
      = it==recDepth
        ?colAppleman
        :new Color(255-5*it%1,255-it%5*30,255-it%5* 50);
    g.setColor(col);
    g.drawLine(x,y,x,y);
  }
```

Schließlich können wir noch die Größe festlegen und das Ganze in einer Hauptmethode starten:

Apfelmaennchen.java

```
  public Dimension getPreferredSize(){
    return new Dimension(width,height);
  }

  public static void main(String [] args){
    JFrame f = new JFrame();
    f.getContentPane().add(new Apfelmaennchen());
    f.pack();
    f.setVisible(true);
  }
}
```

Das Programm ergibt das Bild aus Abbildung 5.4.

5.4 Reaktion auf Ereignisse

Um den graphischen Komponenten eine Funktionalität hinzuzufügen, kennt Java das Konzept der Ereignisbehandlung. Graphische Objekte sollen in der Regel auf bestimmte Ereignisse auf eine definierte Weise reagieren. Solche Ereignisse könne Mausbewegungen, Mausklicks, Ereignisse an einem Fenster, wie das Schließen des Fensters oder etwa Eingaben auf der Tastatur sein. Für die verschiedenen Arten von Ereignissen sind im Paket `java.awt.event` Schnittstellen definiert. In diesen Schnittstellen stehen Methoden, in denen die Reaktion auf bestimmte Ereignisse definiert werden kann. So gibt es z.B. eine Schnittstelle `Mouselistener`, in der Methoden für verschiedene Ereignisse auf den Mausknöpfen bereitstehen.

Soll einer bestimmten graphischen Komponente eine bestimmte Reaktion auf bestimmte Ereignisse zugefügt werden, so ist die entsprechende Schnittstelle mit Methoden für das anvisierte Ereignis ausgeguckt und implementiert werden. Ein Objekt dieser Imple-

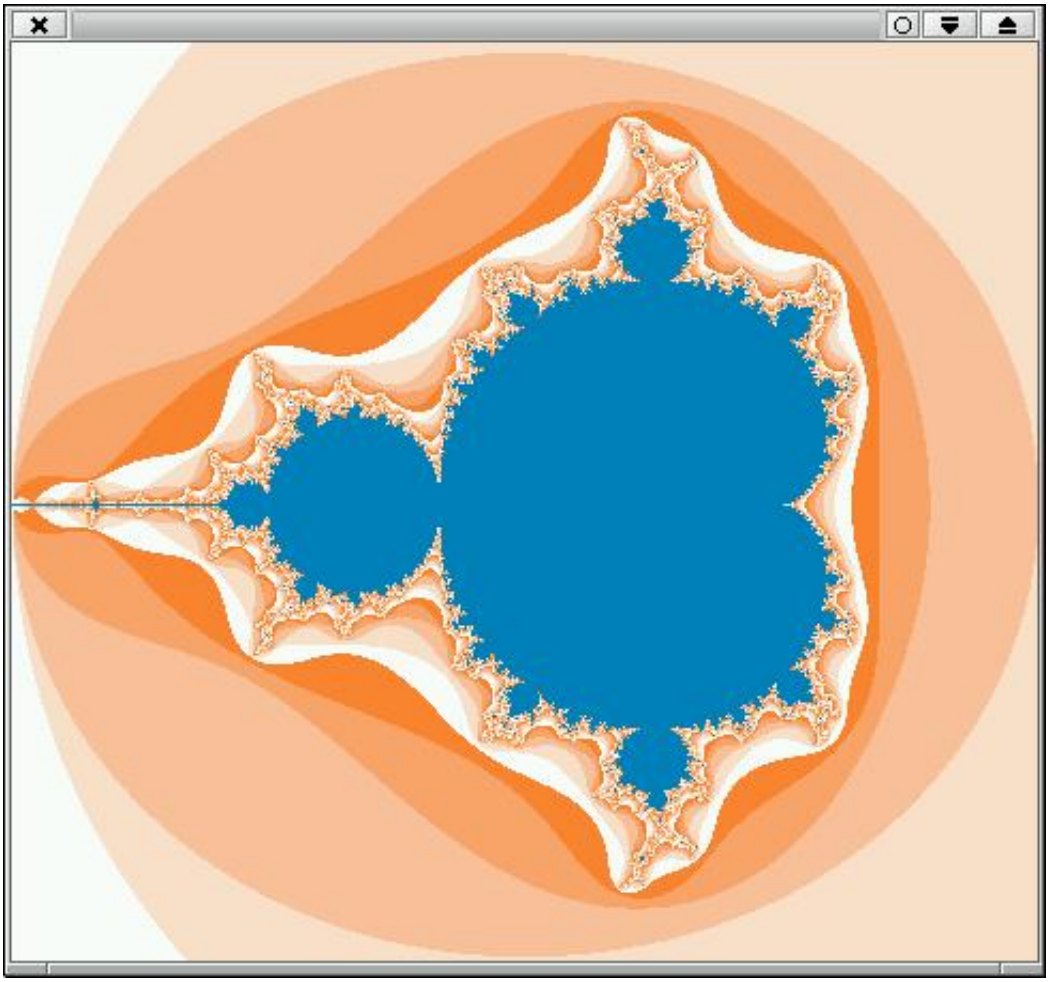

Abbildung 5.4: Das berühmte Apfelmännchen.

mentierung kann dann der graphischen Komponente mit einer entsprechenden Methode hinzugefügt werden.

5.4.1 Der `ActionListener`

Das allgemeinste Ereignis ist ein `ActionEvent`.

Die entsprechende Schnittstelle `ActionListener` enthält nur eine Methode, die auszuführen ist, wenn eine Aktion aufgetreten ist. Dieses Ereignis wird von einem Knopf-Objekt der Klasse `JButton` ausgelöst, wenn ein benutzer auf den Knopf mit der Maus klickt.

Wir implementieren die Schnittstelle `ActionListener` so, dass in einem internen Zähler vermerkt wird, wie oft ein Ereignis aufgetreten ist. Bei jedem Auftreten des Ereignisses

wird die entsprechende Zahl auf einer Textfläche gesetzt:

```
import java.awt.event.*;
import javax.swing.text.*;

class CountActionListener implements ActionListener {
  JTextComponent textArea;
  int count;

  CountActionListener(JTextComponent textArea){
    this.textArea=textArea;
  }

  public void actionPerformed(ActionEvent e) {
    count = count+1;
    textArea.setText(""+count);
  }
}
```

In einer zweiten Klasse definieren wir eine Fensterkomponente mit zwei atomaren Komponenten: einen Knopf und eine Textfläche. Dem Knopf fügen wir die oben geschriebene Ereignisbehandlung hinzu.

```
import java.awt.*;
import javax.swing.*;
import javax.swing.text.*;

class Count extends JFrame {

  public Count(){
    JTextComponent textArea = new JTextField(8);
    JButton button = new JButton("click");
    JPanel pane = new JPanel();
    pane.add(button);
    pane.add(textArea);

    add(pane);
    button.addActionListener(new CountActionListerner(textArea));
    pack();
  }

  public static void main(String [] args){
    JFrame f = new Count();
    f.setVisible(true);
  }
}
```

Wir erhalten ein Fenster, in dem die Anzahl der Mausklicks auf dem Knopf im Textfeld

angezeigt wird.

5.4.2 Innere und Anonyme Klassen

Innere Klassen

Eine innere Klasse wird geschrieben wie jede andere Klasse auch, nur dass sie eben im Rumpf einer äußeren Klasse auftauchen kann. Die innere Klasse hat das Privileg auf die Eigenschaften der äußeren Klasse zuzugreifen, sogar auf die als privat markierten Eigenschaften. Das Attribut privat soll lediglich verhindern, dass eine Eigenschaft von außerhalb der Klasse benutzt wird. Innere Klasse befinden sich aber innerhalb der Klasse.

Unser erstes GUI mit einer Funktionalität lässt sich jetzt mit Hilfe einer inneren Klasse in einer Quelltext-Datei schreiben. Das Feld `counter`, das wir in der vorherigen Implementierung als privates Feld der Klass `CounterListener` definiert hatten, haben wir hier als Feld der GUI-Klasse modelliert. Trotzdem kann die Klasse `CounterListener` weiterhin darauf zugreifen. Ebenso brauch die Textfläche nicht der Klasse `CounterListener` im Konstruktor übergeben werden. Als innere Klasse kann in `CounterListener` auf dieses Feld der äußeren Klasse zugegriffen werden.

InnerCounter.java

```
package name.panitz.simpleGui;
import javax.swing.*;
import java.awt.event.*;

class InnerCounter extends JTB {
  private int counter = 0;

  class CounterListener implements ActionListener{
    public void actionPerformed(ActionEvent ae){
      counter = counter+1;
      textArea.setText(counter+"");
    }
  }

  public InnerCounter(){
    button.addActionListener(new CounterListener());
  }

  public static void main(String [] args){new InnerCounter().showInFrame();}
}
```

Tatsächlich ist die Implementierung kürzer und etwas übersichtlicher geworden.

Beim Übersetzen einer Klasse mit inneren Klassen, erzeugt der Javaübersetzer für jede innere Klasse eine eigene Klassendatei:

```
sep@linux:~/fh/ooseAI/classes/name/panitz/simpleGui> ll *.class
-rw-r--r-- 1 sep u 1082 2014-03-29 11:36 InnerCounter$CounterListener.class
-rw-r--r-- 1 sep u  892 2014-03-29 11:36 InnerCounter.class
```

Der Javaübersetzer schreibt intern den Code um in eine Menge von Klassen ohne innere Klassendefinition und erzeugt für diese den entsprechenden Code. Für die innere Klasse generiert der Javaübersetzer einen Namen, der sich aus äußeren und inneren Klassenamen durch ein Dollarzeichen getrennt zusammensetzt.

Anonyme Klassen

Im letzten Abschnitt hatten wir bereits das Beispiel einer inneren Klasse, für die wir genau einmal ein Objekt erzeugen. In diesem Fall wäre es eigentlich unnötig für eine solche Klasse einen Namen zu erfinden, wenn man an genau dieser einen Stelle, an der das Objekt erzeugt wird, die entsprechende Klasse spezifizieren könnte. Genau hierzu dienen anonyme Klassen in Java. Sie ermöglichen, Klassen ohne Namen zu instanziieren. Hierzu ist nach dem Schlüsselwort `new` anzugeben, von welcher Oberklasse namenlose Klasse ableiten soll, oder welche Schnittstelle mit der namenlosen Klasse implementiert werden soll. Dann folgt nach dem leeren Klammerpaar für dem Konstruktoraufruf in geschweiften Klammern der Rumpf der namenlosen Klasse.

Wir schreiben ein drittes Mal die Klasse `Counter`. Diesmal wird statt der nur einmal instanziierten inneren Klasse eine anonyme Implementierung der Schnittstelle `ActionListener` instanziiert.

AnonymousCounter.java

```
package name.panitz.simpleGui;
import javax.swing.*;
import java.awt.event.*;

class AnonymousCounter extends JTB {
  private int counter = 0;

  public AnonymousCounter(){
    button.addActionListener(
      new ActionListener(){
        public void actionPerformed(ActionEvent e){
          counter = counter+1;
          textArea.setText(counter+"");
        }
      });
  }

  public static void main(String[] a){new AnonymousCounter().showInFrame();}
}
```

Auch für anonyme Klassen generiert der Javaübersetzer eigene Klassendateien. Mangels eines Names, numeriert der Javaübersetzer hierbei die inneren Klassen einfach durch.

```
sep@linux:~/fh/prog2/examples/classes/name/panitz/simpleGui> ll *.class
-rw-r--r-- 1 sep  users  1106 2004-03-29 11:59 AnonymousCounter$1.class
-rw-r--r-- 1 sep  users   887 2004-03-29 11:59 AnonymousCounter.class
sep@linux:~/fh/prog2/examples/classes/name/panitz/simpleGui>
```

5.4.3 Lambda Ausdrücke

Seit Java 8 im Jahre 2014 gibt es eine noch kompaktere Möglichkeit, um einem Knopf die gewünschte Aktion hinzuzufügen. Die Schnittstelle `ActionListener` enthält nur eine einzige Methode. Solche Schnittstelle werden nun als funktionale Schnittstellen bezeichnet, weil sie nur eine Funktion in Form einer Methode enthalten.

Mit Java 8 wurde eine neue Art von Ausdrücken eingeführt. Diese nennen sich Lambda-Ausdrücke. Sie bestehen aus einer Parameterliste in runden Klammern, gefolgt von einem Pfeil, der durch das Minussymbol und das Größersymbol gebildet wird, also `->`. Nach dem Pfeil folgt ein Methodenrumpf oder direkt ein Ausdruck, der zu einem Ergebnis auswertet.

Ein solcher Lambda-Ausdruck ist die Kurzschreibweise für eine Implementierung einer funktionalen Schnittstelle. Es wird also nicht mehr angegeben, welche Schnittstelle implementiert wird, auch nicht mehr, wie die Methode heißt, die implementiert wird, sondern nur noch die Parameterliste und der Methodenrumpf. Alles andere erkennt der Javaübersetzer aus dem Kontext.

Damit wird die kleine Counteranwendung noch kompakter ausgedrückt:

LambdaCounter.java

```
package name.panitz.simpleGui;
import javax.swing.*;
import java.awt.event.*;

class LambdaCounter extends JTB {
  private int counter = 0;

  public LambdaCounter(){
    button.addActionListener( (ev)->{
          counter = counter+1;
          textArea.setText(counter+"");
      });
  }

  public static void main(String [] args){new LambdaCounter().showInFrame();}
}
```

5.4.4 Mausereignisse

Die Schnittstelle `ActionListener` ist dazu geeignet, die Reaktionen einfachsten Ereignisse zu programmieren, den Drücken eines Knopfes.

Ein in modernen graphischen Oberflächen häufigst benutztes Eingabemedium ist die Maus. Zwei verschiedene Ereignisarten sind für die Maus relevant:

- Mausereignisse, die sich auf das Drücken, Freilassen oder Klicken auf einen der Mausknöpfe bezieht. Hierfür gibt es eine Schnittstelle zur Behandlung solcher Ereignisse: `MouseListener`.
- Mausereignisse, die sich auf das Bewegen der Maus beziehen. Die Behandlung solcher Ereignisse kann über eine Implementierung der Schnittstelle `MouseMotionListener` spezifiziert werden.

Entsprechend gibt es für graphische Komponenten Methoden, um solche Mausereignisbehandler der Komponente hinzuzufügen:
`addMouseListener` und `addMouseMotionListener`.

Um die Arbeit mit Ereignisbehandlern zu vereinfachen, gibt es für die entsprechnden Schnittstellen im Paket `java.awt.event` prototypische Implementierungen, in denen die Methoden der Schnittstelle so implementiert sind, daß ohne Aktion auf die entsprechenden Ereignisse reagiert wird. Diese prototypischen Implementierung sind Klassen, deren Namen mit *Adapter* enden. So gibt es zur Schnittstelle `MouseListener` die implementierende Klasse `MouseAdapter`. Will man eine bestimmte Mausbehandlung programmieren, reicht es aus, diesen Adapter zu erweitern und nur die Methoden zu überschreiben, für die bestimmte Aktionen vorgesehen sind. Es erübrigt sich dann für alle sechs Methoden der Schnittstelle `MouseListener` Implementierungen vorzusehen.

Wir erweitern die Klasse `Apfelmaennchen` um eine Mausbehandlung. Der mit gedrückter Maus markierte Bereich soll vergrößert in dem Fenster dargestellt werden.

ApfelWithMouse.java

```
package name.panitz.crempel.tool.apfel;

import java.awt.Graphics;
import java.awt.event.*;
import javax.swing.JFrame;

public class ApfelWithMouse extends Apfelmaennchen{
  public ApfelWithMouse(){
```

Im Konstruktor fügen wir der Komponente eine Mausbehandlung hinzu. Der Mausbehandler merkt sich die Koordinaten, an denen die Maus gedrückt wird und berechnet beim Loslassen des Mausknopfes den neuen darzustellenden Zahlenbereich:

ApfelWithMouse.java

```
    addMouseListener(new MouseAdapter(){
      int mouseStartX=0;
      int mouseStartY=0;

      public void mousePressed(MouseEvent e) {
        mouseStartX=e.getX();
        mouseStartY=e.getY();
      }

      public void mouseReleased(MouseEvent e) {
        int endX = e.getX();
        int endY = e.getY();
        startX = startX+(mouseStartX*zelle);
        startY = startY+(mouseStartY*zelle);
        zelle  = zelle*(endX-mouseStartX)/width;
        repaint();
      }
    });
  }
```

Auch für diese Klasse sehen wir eine kleine Startmethode vor:

ApfelWithMouse.java

```
  public static void main(String [] args){
    JFrame f = new JFrame();
    f.getContentPane().add(new ApfelWithMouse());
    f.pack();
    f.setVisible(true);
  }
}
```

5.4.5 Fensterereignisse

Auch für Fenster in einer graphischen Benutzeroberfläche existieren eine Reihe von Ereignissen. Das Fenster kann minimiert oder maximiert werden, es kann das aktive Fenster oder im Hintergrund sein und es kann schließlich auch geschlossen werden. Um die Reaktion auf solche Ereignisse zu spezifizieren existiert die Schnittstelle `WindowListener` mit entsprechender prototypischer Adapterklasse `WindowAdapter`. Die Objekte der Fenstereignisbehandlung können mit der Methode `addWindowListener` Fensterkomponenten hinzugefügt werden.

In den bisher vorgestellten Programmen wird Java nicht beendet, wenn das einzige Fenster der Anwendung geschlossen wurde. Man kann an der Konsole sehen, dass der Javainterpreter weiterhin aktiv ist. Das liegt daran, dass wir bisher noch nicht spezifiziert

haben, wie die Fensterkomponenten auf das Ereignis des Schließens des Fensters reagieren sollen. Dieses kann mit einem Objekt, das `WindowListener` implementiert in der Methode `windowClosing` spezifiziert werden. Wir schreiben hier eine Version des Apfelmännchenprogramms, in dem das Schließen des Fensters den Abbruch des gesamten Programms bewirkt.

ClosingApfelFrame.java

```
package name.panitz.crempel.tool.apfel;

import javax.swing.JFrame;
import java.awt.event.*;

public class ClosingApfelFrame {
  public static void main(String [] args){
    JFrame f = new JFrame();
    f.add(new ApfelWithMouse());
    f.addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
    f.pack();
    f.setVisible(true);
  }
}
```

5.5 Zeitgesteuerte Ereignisse

Um zeitlich immer wiederkehrende Ereignisse in GUIs zu programmieren gibt in Swing eine Hilfsklasse `Timer`. Objekte dieser Klasse können so instanziiert werden, dass sie in bestimmten Zeitabständen Ereignisse auslösen. Der `Timer` ist also so etwas wie ein Ereignisgenerator. Zusätzlich gibt man einem `Timer`-Objekt auch einen `ActionListener` mit, der spezifiziert, wie auf diese in Zeitintervallen auftretenden Ereignisse reagiert werden soll.

Folgende Klasse implementiert eine simple Uhr. In einem `JLabel` wird die aktuelle Zeit angegeben. Die Komponente wird einem `Timer` übergeben, der jede Sekunde eine neues Ereignis erzeugt. Diese Ereignisse sorgen dafür, dass die Zeit im Label aktualisiert wird.

Uhr.java

```
package name.panitz.oose.swing.examples;

import javax.swing.*;
import java.util.Date;
import java.awt.event.*;
```

Die Klasse Uhr ist nicht nur ein `JPanel`, in dem ein `JLabel` benutzt wird, Datum und Uhrzeit anzuzeigen, sondern implementiert gleichfalls auch einen `ActionListener`.

Uhr.java

```
public class Uhr extends JPanel implements ActionListener{
```

Zunächst sehen wir das Datumsfeld für diese Komponente vor:

Uhr.java

```
  JLabel l = new JLabel(new Date()+"");
```

Im Konstruktor erzeugen wir ein Objekt vom Typ `Timer`. Dieses Objekt soll alle Sekunde (alle 1000 Millisekunden) ein Ereignis erzeugen. Dem `Timer` wird das gerade im Konstruktor erzeugte Objekt vom Typ `Uhr` übergeben, das, da es ja einen `ActionListener` implementiert, auf diese Ereignisse reagieren soll.

Uhr.java

```
  public Uhr (){
    new Timer(1000,this).start();
    add(l);
  }
```

Um die Schnittstelle `ActionListener` korrekt zu implementieren, muss die Methode `actionPerformed` implementiert werden. In dieser setzen wir jeweils Datum und Uhrzeit mit dem aktuellen Wert neu ins Label.

Uhr.java

```
  public void actionPerformed(ActionEvent ae){
    l.setText(""+new Date());
  }
```

Und natürlich sehen wir zum Testen eine kleine Hauptmethode vor, die die Uhr in einem Fensterrahmen anzeigt.

Uhr.java

```
  public static void main(String [] args){
    JFrame f = new JFrame();
    f.getContentPane().add(new Uhr());
    f.pack();
    f.setVisible(true);
  }
}
```

5.5.1 Animationen

Mit dem Prinzip des Timers können wir jetzt auf einfache Weise Animationen realisieren. In einer Animation bewegt sich etwas. Dieses drücken wir durch eine entsprechende Schnittstelle aus:

Animation.java

```
package name.panitz.animation;

public interface Animation {
  public void move();
}
```

Wir wollen einen besonderen `JPanel` realisieren, in dem sich etwas bewegen kann. Damit soll ein solcher `JPanel` auch eine Animation sein. Es bietet sich an, eine abstrakte Klasse zu schreiben, in der die Methode `move` noch nicht implementiert ist:

AnimatedJPanel.java

```
package name.panitz.animation;

import javax.swing.JPanel;
import javax.swing.Timer;
import java.awt.event.*;

public abstract class AnimatedJPanel
  extends JPanel implements Animation {
```

Um zeitgesteuert das Szenario der Animation zu verändern, brauchen wir einen `Timer`.

AnimatedJPanel.java

```
  Timer t;
```

Im Konstruktor wird dieser initialisiert. Als Ereignisbehandlung wird ein Ereignisbehandlungsobjekt erzeugt, das die Methode `move` aufruft, also dafür sorgt, dass die Szenerie

sich weiterbewegt und das dafür sorgt, dass die Szenerie neu gezeichnet wird. Wir starten diesen Timer gleich.

AnimatedJPanel.java

```
  public AnimatedJPanel(){
    super(true);
    t = new Timer(29,new ActionListener(){
      public void actionPerformed(ActionEvent ae){
        move();
        repaint();
      }
    });
    t.start();
  }
}
```

Jetzt können wir durch implementieren der Methode `move` und Überschreiben der Methode `paintComponent` beliebige Animationen erzeugen. Als erstes schreiben wir eine Klasse in der ein Kreis sich auf und ab bewegt:

BouncingBall.java

```
package name.panitz.animation;

import java.awt.Graphics;
import java.awt.Dimension;
import java.awt.Color;
import javax.swing.JFrame;

public class BouncingBall extends AnimatedJPanel {
```

Die Größe des Kreises und des Spielfeldes setzen wir in Konstanten fest:

BouncingBall.java

```
  final int width = 100;
  final int height = 200;
  final int ballSize = 20;
```

Der Ball soll sich entlang der y-Achse bewegen, und zwar pro Bild um 4 Pixel:

BouncingBall.java

```
  int yDir = 4;
```

Anfangs soll der Ball auf der Hälfte der x-Achse liegen und ganz oben im Bild liegen:

BouncingBall.java

```
  int ballX = width/2-ballSize/2;
  int ballY = 0;
```

Wir bewegen den Ball. Wenn er oben oder unten am Spielfeldrand anstößt, so ändert er seine Richtung:

BouncingBall.java

```
  public void move(){
    if (ballY>height-ballSize || ballY<0) yDir=-yDir;
    ballY=ballY+yDir;
  }
```

Zum Zeichnen, wird ein roter Hintergrund gezeichnet und der Kreis an seiner aktuellen Position.

BouncingBall.java

```
  public void paintComponent(Graphics g){
    g.setColor(Color.RED);
    g.fillRect(0,0,width,height);
    g.setColor(Color.YELLOW);
    g.fillOval(ballX,ballY,ballSize,ballSize);
  }
```

Unsere Größe wird verwendet als bevorzugte Größe der Komponente:

BouncingBall.java

```
  public Dimension getPreferredSize(){
    return new Dimension(width,height);
  }
```

Und schließlich folgt eine kleine Hauptmethode zum Starten der Animation.

BouncingBall.java

```
  public static void main(String [] args){
    JFrame f = new JFrame("");
    f.add(new BouncingBall());
    f.pack();
    f.setVisible(true);
  }
}
```

5.6 Weitere Swing Komponenten

Um einen kleinen Überblick der vorhandenen Swing Komponenten zu bekommen, können wir ein kleines Programm schreiben, daß möglichst viele Komponenten einmal instanziiert. Hierzu schreiben wir eine kleine Testklasse:

ComponentOverview.java

```
package name.panitz.gui.example;

import javax.swing.*;
import java.awt.*;
import java.util.*;

public class ComponentOverview  {
  public ComponentOverview(){
```

Darin definieren wir eine Reihung von einfachen Swing-Komponenten:

ComponentOverview.java

```
    JComponent [] cs1 =
     {new JButton("knopf")
     ,new JCheckBox("check mich")
     ,new JRadioButton("drück mich")
     ,new JMenuItem("ins Menue mit mir")
     ,new JComboBox(combos)
     ,new JList(combos)
     ,new JSlider(0,350,79)
     ,new JSpinner(new SpinnerNumberModel(18,0.0,42.0,2.0))
     ,new JTextField(12)
     ,new JFormattedTextField("hallo")
     ,new JLabel("einfach nur ein Label")
     ,new JProgressBar(0,42)
     };
```

Sowie eine zweite Reihung von komplexeren Swing-Komponenten:

ComponentOverview.java

```
    JComponent [] cs2 =
     {new JColorChooser(Color.RED)
     ,new JFileChooser()
     ,new JTable(13,5)
     ,new JTree()
     };
```

Diese beiden Reihungen zeigen sollen mit einer Hilfsmethode in einem Fenster angezeigt werden:

ComponentOverview.java

```
 displayComponents(cs1,3);
 displayComponents(cs2,2);
}
```

Für die Listen- und Auswahlkomponenten oben haben wir eine Reihung von Strings benutzt:

ComponentOverview.java

```
String [] combos = {"friends","romans","contrymen"};
```

Bleibt die Methode zu schreiben, die die Reihungen von Komponenten anzeigen kann. Als zweites Argument bekommt diese Methode übergeben, in wieviel Spalten die Komponenten angezeigt werden sollen.

ComponentOverview.java

```
public void displayComponents(JComponent [] cs,int col){
```

Ein Fenster wird definiert, für das eine Gridlayout-Zwischenkomponente mit genügend Zeilen erzeugt wird:

ComponentOverview.java

```
  JFrame f = new JFrame();
  JPanel panel = new JPanel();
  panel.setLayout(
   new GridLayout(cs.length/col+(cs.length%col==0?0:1),col));
```

Für jede Komponente wird ein `Panel` mit Rahmen und den Klassennamen der Komponente als Titel erzeugt und der Zwischenkomponente hinzugefügt:

ComponentOverview.java

```
  for (JComponent c:cs){
    JPanel p = new JPanel();
    p.add(c);
    p.setBorder(BorderFactory
                .createTitledBorder(c.getClass().getName()));
    panel.add(p);
  }
```

Schließlich wird noch das Hauptfester zusammengepackt:

ComponentOverview.java

```
    f.getContentPane().add(panel);
    f.pack();
    f.setVisible(true);
  }
```

Und um alles zu starten, noch eine kleine Hauptmethode:

ComponentOverview.java

```
  public static void main(String [] args){
    new  ComponentOverview();
  }
}
```

Wir erhalten einmal die Übersicht von Komponenten wie in Abbildung 5.5 und einmal wie in Abbildung 5.6 dargestellt.

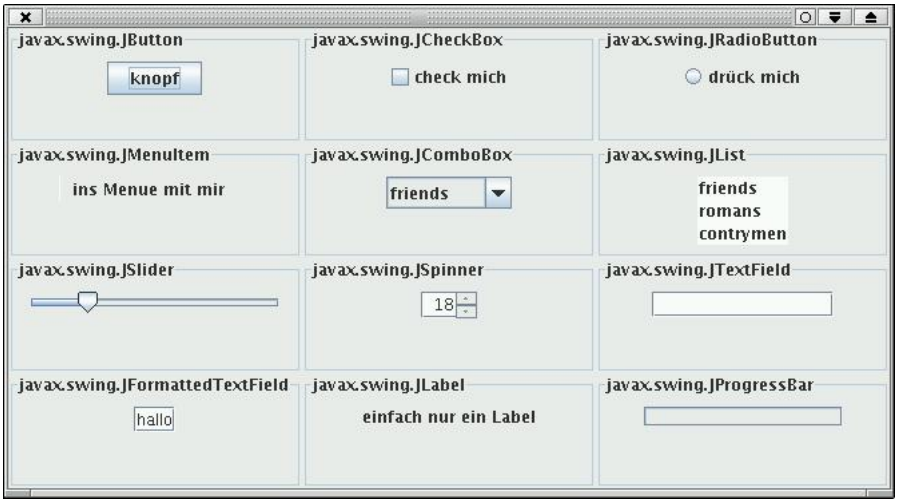

Abbildung 5.5: Überblick über einfache Komponenten.

Womit wir uns noch nicht im einzelnen beschäftigt haben, ist das Datenmodell, das hinter den einzelnen komplexeren Komponenten steht.

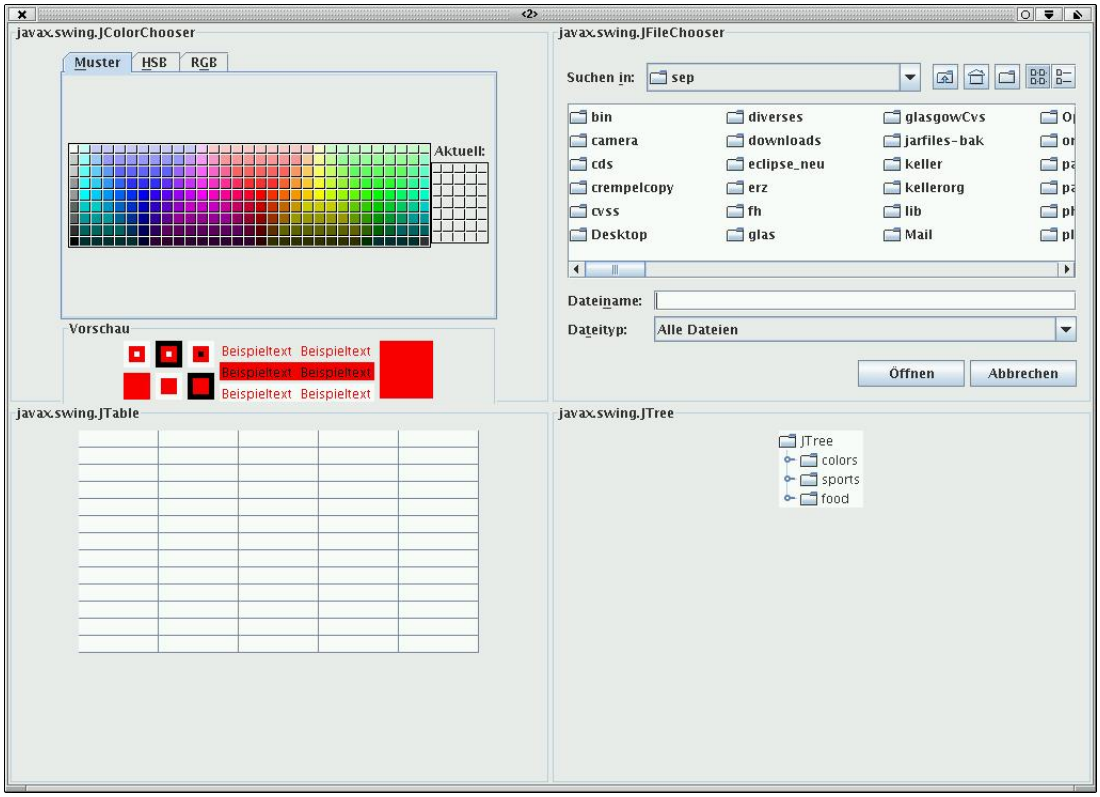

Abbildung 5.6: Überblick über komplexere Komponenten.

6 Weiterführende Konzepte

6.1 Aufzählungstypen

Ein erster Aufzählungstyp für die Wochentage.

```
package name.panitz.enums;
public enum Wochentage {
  montag,dienstag,mittwoch,donnerstag
 ,freitag,sonnabend,sonntag;
}
```

Auch dieses neue Konstrukt wird von Javaübersetzer in eine herkömmlige Javaklasse übersetzt. Wir können uns davon überzeugen, indem wir uns einmal den Inhalt der erzeugten Klassendatei mit `javap` wieder anzeigen lassen:

```
linux:fh/> javap name.panitz.enums.Wochentage
Compiled from "Wochentage.java"
public class name.panitz.enums.Wochentage extends java.lang.Enum{
    public static final name.panitz.enums.Wochentage montag;
    public static final name.panitz.enums.Wochentage dienstag;
    public static final name.panitz.enums.Wochentage mittwoch;
    public static final name.panitz.enums.Wochentage donnerstag;
    public static final name.panitz.enums.Wochentage freitag;
    public static final name.panitz.enums.Wochentage sonnabend;
    public static final name.panitz.enums.Wochentage sonntag;
    public static final name.panitz.enums.Wochentage[] values();
    public static name.panitz.enums.Wochentage valueOf(java.lang.String);
    public name.panitz.enums.Wochentage(java.lang.String, int);
    public int compareTo(java.lang.Enum);
    public int compareTo(java.lang.Object);
    static {};
}
```

Eine der schönen Eigenschaften der Aufzählungstypen ist, daß sie in einer `switch`-Anweisung benutzt werden können.

Wir fügen der Aufzählungsklasse eine Methode zu, um zu testen ob der Tag ein Werktag ist. Hierbei läßt sich eine `switch`-Anweisung benutzen.

```
package name.panitz.enums;
public enum Tage {
  montag,dienstag,mittwoch,donnerstag
 ,freitag,sonnabend,sonntag;

  public boolean isWerktag(){
    switch (this){
      case sonntag    :
      case sonnabend  :return false;
      default         :return true;
    }
  }

  public static void main(String [] _){
    Tage tag = freitag;
    System.out.println(tag);
    System.out.println(tag.ordinal());
    System.out.println(tag.isWerktag());
    System.out.println(sonntag.isWerktag());
  }
}
```

Das Programm gibt die erwartete Ausgabe:

```
linux:~/> java -classpath classes/ name.panitz.enums.Tage
freitag
4
true
false
```

Eine angenehme Eigenschaft der Aufzählungsklassen ist, daß sie in einer Reihung alle Werte der Aufzählung enthalten, so daß mit der neuen `for`-Schleife bequem über diese iteriert werden kann.

Wir iterieren in diesem Beispiel einmal über alle Wochentage.

```
package name.panitz.enums;
public class IterTage {
  public static void main(String [] _){
    for (Tage tag:Tage.values())
      System.out.println(tag.ordinal()+": "+tag);
  }
}
```

Die erwarttete Ausgabe ist:

```
linux:~/> java -classpath classes/ name.panitz.enums.IterTage
0: montag
1: dienstag
2: mittwoch
3: donnerstag
4: freitag
5: sonnabend
6: sonntag
```

Schließlich kann man den einzelnen Konstanten einer Aufzählung noch Werte übergeben.

Wir schreiben eine Aufzählung für die Euroscheine. Jeder Scheinkonstante wird noch eine ganze Zahl mit übergeben. Es muß hierfür ein allgemeiner Konstruktor geschrieben werden, der diesen Parameter übergeben bekommt.

```
package name.panitz.enums;
public enum Euroschein {
   fünf(5),zehn(10),zwanzig(20),fünfzig(50),hundert(100)
  ,zweihundert(200),fünfhundert(500);
  private int value;
  public Euroschein(int v){value=v;}
  public int value(){return value();}

  public static void main(String [] _){
    for (Euroschein schein:Euroschein.values())
      System.out.println
        (schein.ordinal()+": "+schein+" -> "+schein.value);
  }
}
```

Das Programm hat die folgende Ausgabe:

```
linux:~/> java -classpath classes/ name.panitz.enums.Euroschein
0: fünf -> 5
1: zehn -> 10
2: zwanzig -> 20
3: fünfzig -> 50
4: hundert -> 100
5: zweihundert -> 200
6: fünfhundert -> 500
```

6.2 Variable Parameteranzahl

Als zusätzliches kleines Gimmik ist in Java 1.5 eingebaut worden, daß Methoden mit einer variablen Parameteranzahl definiert werden können. Dieses wird durch drei Punkte

nach dem Parametertyp in der Signatur gekennzeichnet. Damit wird angegeben, daß eine bieliebige Anzahl dieser Parameter bei einem Methodenaufruf geben kann.

Es läßt sich so eine Methode schreiben, die mit beliebig vielen Stringparametern aufgerufen werden kann.

```
package name.panitz.java15;

public class VarParams{
  static public String append(String... args){
    String result="";
    for (String a:args)
      result=result+a;
    return result;
  }

  public static void main(String [] _){
    System.out.println(append("hello"," ","world"));
  }
}
```

Die Methode `append` konkateniert endlich viele `String`-Objekte.

Wie schon für Aufzählungen können wir auch einmal schauen, was für Code der Javakompilierer für solche Methoden erzeugt.

```
inux:~/> javap -classpath classes/ name.panitz.java15.VarParams
Compiled from "VarParams.java"
public class name.panitz.java15.VarParams extends java.lang.Object{
    public name.panitz.java15.VarParams();
    public static java.lang.String append(java.lang.String[]);
    public static void main(java.lang.String[]);
}
```

Wie man sieht wird für die variable Parameteranzahl eine Reihung erzeugt. Der Javakompilierer sorgt bei Aufrufen der Methode dafür, daß die entsprechenden Parameter in eine Reihung verpackt werden. Daher können wir mit dem Parameter wie mit einer Reihung arbeiten.

6.3 Generische Typen

Generische Typen wurden im JSR014 definiert. In der Expertengruppe des JSR014 war der Autor dieses Skripts zeitweilig als Stellvertreter der Software AG Mitglied. Die Software AG hatte mit der Programmiersprache Bolero bereits einen Compiler für generische Typen implementiert. Der Bolero Compiler generiert auch Java Byte Code. Von dem ersten Wunsch nach Generizität bis zur nun vorliegenden Javaversion 1.5 sind viele Jahre vergangen. Andere wichtige JSRs, die in Java 1.5 integriert werden, tragen bereits die

Nummern 175 und 201. Hieran kann man schon erkennen, wie lange es gedauert hat, bis generische Typen in Java integriert wurden.

Interessierten Programmierern steht schon seit Mitte der 90er Jahre eine Javaerweiterung mit generischen Typen zur Verfügung. Unter den Namen Pizza existiert eine Javaerweiterung, die nicht nur generische Typen, sondern auch algebraische Datentypen mit *pattern matching* und Funktionsobjekten zu Java hinzufügte. Unter den Namen *GJ* für *Generic Java* wurde eine allein auf generische Typen abgespeckte Version von *Pizza* publiziert. *GJ* ist tatsächlich der direkte Prototyp für Javas generische Typen. Die Expertenrunde des JSR014 hat *GJ* als Grundlage für die Spezifikation genommen und an den grundlegenden Prinzipien auch nichts mehr geändert.

6.3.1 Generische Klassen

Die Idee für generische Typen ist, eine Klasse zu schreiben, die für verschiedene Typen als Inhalt zu benutzen ist. Das geht bisher in Java, allerdings mit einem kleinen Nachteil. Versuchen wir einmal, in traditionellem Java eine Klasse zu schreiben, in der wir beliebige Objekte speichern können. Um beliebige Objekte speichern zu können, brauchen wir ein Feld, in dem Objekte jeden Typs gespeichert werden können. Dieses Feld muss daher den Typ `Object` erhalten:

OldBox.java

```
class OldBox {
  Object contents;
  OldBox(Object contents){this.contents=contents;}
}
```

Der Typ `Object` ist ein sehr unschöner Typ; denn mit ihm verlieren wir jegliche statische Typinformation. Wenn wir die Objekte der Klasse `OldBox` benutzen wollen, so verlieren wir sämtliche Typinformation über das in dieser Klasse abgespeicherte Objekt. Wenn wir auf das Feld `contents` zugreifen, so haben wir über das darin gespeicherte Objekte keine spezifische Information mehr. Um das Objekt weiter sinnvoll nutzen zu können, ist eine dynamische Typzusicherung durchzuführen:

UseOldBox.java

```
class UseOldBox{
  public static void main(String [] args){
    OldBox b = new OldBox("hello");
    String s = (String)b.contents;
    System.out.println(s.toUpperCase());
    System.out.println(((String) s).toUpperCase());
  }
}
```

Wann immer wir mit dem Inhalt des Felds `contents` arbeiten wollen, ist die Typzusicherung während der Laufzeit durchzuführen. Die dynamische Typzusicherung kann zu einem Laufzeitfehler führen. So übersetzt das folgende Programm fehlerfrei, ergibt aber einen Laufzeitfehler:

UseOldBoxError.java

```
class UseOldBoxError{
  public static void main(String [] args){
    OldBox b = new OldBox(new Integer(42));
    String s = (String)b.contents;
    System.out.println(s.toUpperCase());
  }
}
```

```
sep@linux:~/fh/java1.5/examples/src> javac UseOldBoxError.java
sep@linux:~/fh/java1.5/examples/src> java UseOldBoxError
Exception in thread "main" java.lang.ClassCastException
        at UseOldBoxError.main(UseOldBoxError.java:4)
sep@linux:~/fh/java1.5/examples/src>
```

Wie man sieht, verlieren wir Typsicherheit, sobald der Typ `Object` benutzt wird. Bestimmte Typfehler können nicht mehr statisch zur Übersetzungszeit, sondern erst dynamisch zur Laufzeit entdeckt werden.

Der Wunsch ist, Klassen zu schreiben, die genauso allgemein benutzbar sind wie die Klasse `OldBox` oben, aber trotzdem die statische Typsicherheit garantieren, indem sie nicht mit dem allgemeinen Typ `Object` arbeiten. Genau dieses leisten generische Klassen. Hierzu ersetzen wir in der obigen Klasse jedes Auftreten des Typs `Object` durch einen Variablennamen. Diese Variable ist eine Typvariable. Sie steht für einen beliebigen Typen. Dem Klassennamen fügen wir zusätzlich in der Klassendefinition in spitzen Klammern eingeschlossen hinzu, daß diese Klasse eine Typvariable benutzt. Wir erhalten somit aus der obigen Klasse `OldBox` folgende generische Klasse `Box`.

Box.java

```
class Box<ET> {
  ET contents;
  Box(ET  contents){this.contents=contents;}
}
```

Die Typvariable `ET` ist als allquantifiziert zu verstehen. Für jeden Typ `ET` können wir die Klasse `Box` benutzen. Man kann sich unsere Klasse `Box` analog zu einer realen Schachtel vorstellen: Beliebige Dinge können in die Schachtel gelegt werden. Betrachten wir dann allein die Schachtel von außen, können wir nicht mehr wissen, was für ein Objekt darin enthalten ist. Wenn wir viele Dinge in Schachteln packen, dann schreiben wir auf die Schachtel jeweils drauf, was in der entsprechenden Schachtel enthalten ist. Ansons-

ten würden wir schnell die Übersicht verlieren. Und genau das ermöglichen generische Klassen. Sobald wir ein konkretes Objekt der Klasse `Box` erzeugen wollen, müssen wir entscheiden, für welchen Inhalt wir eine Box brauchen. Dieses geschieht, indem in spitzen Klammern dem Klassennamen `Box` ein entsprechender Typ für den Inhalt angehängt wird. Wir erhalten dann z.B. den Typ `Box<String>`, um Strings in der Schachtel zu speichern, oder `Box<Integer>`, um Integerobjekte darin zu speichern:

UseBox.java

```
class UseBox{
  public static void main(String [] args){
    Box<String> b1 = new Box<String>("hello");
    String s = b1.contents;
    System.out.println(s.toUpperCase());
    System.out.println(b1.contents.toUpperCase());

    Box<Integer> b2 = new Box<Integer>(new Integer(42));

    System.out.println(b2.contents.intValue());
  }
}
```

Wie man im obigen Beispiel sieht, fallen jetzt die dynamischen Typzusicherungen weg. Die Variablen `b1` und `b2` sind jetzt nicht einfach vom Typ `Box`, sondern vom Typ `Box<String>` respektive `Box<Integer>`.

Da wir mit generischen Typen keine Typzusicherungen mehr vorzunehmen brauchen, bekommen wir auch keine dynamischen Typfehler mehr. Der Laufzeitfehler, wie wir ihn ohne die generische Box hatten, wird jetzt bereits zur Übersetzungszeit entdeckt. Hierzu betrachte man das analoge Programm:

Fehlerhafter Code!

```
class UseBoxError{
  public static void main(String [] args){
    Box<String> b = new Box<String>(new Integer(42));
    String s = b.contents;
    System.out.println(s.toUpperCase());
  }
}
```

Die Übersetzung dieses Programms führt jetzt bereits zu einen statischen Typfehler:

```
sep@linux:~/fh/java1.5/examples/src> javac UseBoxError.java
UseBoxError.java:3: cannot find symbol
symbol  : constructor Box(java.lang.Integer)
location: class Box<java.lang.String>
    Box<String> b = new Box<String>(new Integer(42));
                    ^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

6.3.2 Vererbung

Generische Typen sind ein Konzept, das orthogonal zur Objektorientierung ist. Von generischen Klassen lassen sich in gewohnter Weise Unterklassen definieren. Diese Unterklassen können, aber müssen nicht selbst generische Klassen sein. So können wir unsere einfache Schachtelklasse erweitern, so dass wir zwei Objekte speichern können:

GPair.java

```
class GPair<A,B> extends Box<A>{
  GPair(A x,B y){
    super(x);
    snd = y;
  }

  B snd;

  public String toString(){
    return "("+contents+","+snd+")";
  }
}
```

Die Klasse `GPair` hat zwei Typvariablen. Instanzen von `GPair` müssen angeben von welchem Typ die beiden zu speichernden Objekte sein sollen.

UsePair.java

```
class UsePair{
  public static void main(String [] args){
    GPair<String,Integer> p
     = new GPair<String,Integer>("hallo",new  Integer(40));

    System.out.println(p);
    System.out.println(p.contents.toUpperCase());
    System.out.println(p.snd.intValue()+2);
  }
}
```

Wie man sieht kommen wir wieder ohne Typzusicherung aus. Es gibt keinen dynamischen Typcheck, der im Zweifelsfall zu einer Ausnahme führen könnte.

```
sep@linux:~/fh/java1.5/examples/classes> java UsePair
(hallo,40)
HALLO
42
sep@linux:~/fh/java1.5/examples/classes>
```

Wir können auch eine Unterklasse bilden, indem wir mehrere Typvariablen zusammenfassen. Wenn wir uniforme Paare haben wollen, die zwei Objekte gleichen Typs speichern, können wir hierfür eine spezielle Paarklasse definieren.

UniPair.java

```
class UniPair<A> extends GPair<A,A>{
  UniPair(A x,A y){super(x,y);}
  void swap(){
    final A z = snd;
    snd = contents;
    contents = z;
  }
}
```

Da beide gespeicherten Objekte jeweils vom gleichen Typ sind, konnten wir jetzt eine Methode schreiben, in der diese beiden Objekte ihren Platz tauschen. Wie man sieht, sind Typvariablen ebenso wie unsere bisherigen Typen zu benutzen. Sie können als Typ für lokale Variablen oder Parameter genutzt werden.

UseUniPair.java

```
class UseUniPair{
  public static void main(String [] args){
    UniPair<String> p
     = new UniPair<String>("welt","hallo");

    System.out.println(p);
    p.swap();
    System.out.println(p);
  }
}
```

Wie man bei der Benutzung der uniformen Paare sieht, gibt man jetzt natürlich nur noch einen konkreten Typ für die Typvariablen an. Die Klasse `UniPair` hat ja nur eine Typvariable.

```
sep@linux:~/fh/java1.5/examples/classes> java UseUniPair
(welt,hallo)
(hallo,welt)
sep@linux:~/fh/java1.5/examples/classes>
```

Wir können aber auch Unterklassen einer generischen Klasse bilden, die nicht mehr generisch ist. Dann leiten wir für eine ganz spezifische Instanz der Oberklasse ab. So läßt sich z.B. die Klasse `Box` zu einer Klasse erweitern, in der nur noch Stringobjekte verpackt werden können:

StringBox.java

```
class StringBox extends Box<String>{
  StringBox(String x){super(x);}
}
```

Diese Klasse kann nun vollkommen ohne spitze Klammern benutzt werden:

UseStringBox.java

```
class UseStringBox{
  public static void main(String [] args){
    StringBox b = new StringBox("hallo");
    System.out.println(b.contents.length());
  }
}
```

6.3.3 Generische Schnittstellen

Generische Typen erlauben es, den Typ `Object` in Typsignaturen zu eleminieren. Der Typ `Object` ist als schlecht anzusehen, denn er ist gleichbedeutend damit, daß keine Information über einen konkreten Typ während der Übersetzungszeit zur Verfügung steht. Im herkömmlichen Java ist in APIs von Bibliotheken der Typ `Object` allgegenwärtig. Sogar in der Klasse `Object` selbst begegnet er uns in Signaturen. Die Methode `equals` hat einen Parameter vom Typ `Object`, d.h. prinzipiell kann ein Objekt mit Objekten jeden beliebigen Typs verglichen werden. Zumeist will man aber nur gleiche Typen miteinander vergleichen. In diesem Abschnitt werden wir sehen, daß generische Typen es uns erlauben, allgemein eine Gleichheitsmethode zu definieren, in der nur Objekte gleichen Typs miteinander verglichen werden können. Hierzu werden wir eine generische Schnittstelle definieren.

Generische Typen erweitern sich ohne Umstände auf Schnittstellen. Im Vergleich zu generischen Klassen ist nichts Neues zu lernen. Syntax und Benutzung funktionieren auf die gleiche Weise.

Äpfel mit Birnen vergleichen

Um zu realisieren, daß nur noch Objekte gleichen Typs miteinander verglichen werden können, definieren wir eine Gleichheitsschnitstelle. In ihr wird eine Methode spezifiziert, die für die Gleichheit stehen soll. Die Schnittstelle ist generisch über den Typen, mit dem vergleichen werden soll.

EQ.java

```
interface EQ<E> {
  public boolean eq(E that);
}
```

Jetzt können wir für jede Klasse nicht nur bestimmen, dass sie die Gleichheit implementieren soll, sondern auch, mit welchen Typen Objekte unserer Klasse verglichen werden sollen. Schreiben wir hierzu eine Klasse `Apfel`. Die Klasse Apfel soll die Gleichheit auf sich selbst implementieren. Wir wollen nur Äpfel mit Äpfeln vergleichen können. Daher definieren wir in der `implements`-Klausel, dass wir `EQ<Apfel>` implementieren wollen. Dann müssen wir auch die Methode `eq` implementieren, und zwar mit dem Typ `Apfel` als Parametertyp:

Apfel.java

```
class Apfel implements EQ<Apfel>{
  String typ;

  Apfel(String typ){
    this.typ=typ;}

  public boolean eq(Apfel that){
    return this.typ.equals(that.typ);
  }
}
```

Jetzt können wir Äpfel mit Äpfeln vergleichen:

TestEQ.java

```
class TestEq{
  public static void main(String []_){
    Apfel a1 = new Apfel("Golden Delicious");
    Apfel a2 = new Apfel("Macintosh");
    System.out.println(a1.eq(a2));
    System.out.println(a1.eq(a1));
  }
}
```

Schreiben wir als nächstes eine Klasse die Birnen darstellen soll. Auch diese implementiere die Schnittstelle `EQ`, und zwar dieses Mal für Birnen:

Birne.java

```
class Birne implements EQ<Birne>{
  String typ;
  Birne(String typ){
    this.typ=typ;}

  public boolean eq(Birne that){
    return this.typ.equals(that.typ);
  }
}
```

Während des statischen Typchecks wird überprüft, ob wir nur Äpfel mit Äpfeln und Birnen mit Birnen vergleichen. Der Versuch, Äpfel mit Birnen zu vergleichen, führt zu einem Typfehler:

Fehlerhafter Code!

```
class TesteEqError{
  public static void main(String []_){
    Apfel a = new Apfel("Golden Delicious");
    Birne b = new Birne("williams");
    System.out.println(a.equals(b));
    System.out.println(a.eq(b));
  }
}
```

Wir bekommen die verständliche Fehlermeldung, dass die Gleichheit auf Äpfel nicht für einen Birnenparameter aufgerufen werden kann.

```
./TestEQError.java:6: eq(Apfel) in Apfel cannot be applied to (Birne)
    System.out.println(a.eq(b));
                       ^
1 error
```

Wahrscheinlich ist es jedem erfahrenden Javaprogrammierer schon einmal passiert, dass er zwei Objekte mit der Methode `equals` verglichen hat, die er gar nicht vergleichen wollte. Da der statische Typcheck solche Fehler nicht erkennen kann, denn die Methode `equals` lässt jedes Objekt als Parameter zu, sind solche Fehler mitunter schwer zu lokalisieren.

Der statische Typcheck stellt auch sicher, dass eine generische Schnittstelle mit der korrekten Signatur implementiert wird. Der Versuch, eine Birnenklasse zu schreiben, die eine Gleichheit mit Äpfeln implementieren soll, dann aber die Methode `eq` mit dem Parametertyp `Birne` zu implementieren, führt ebenfalls zu einer Fehlermeldung:

Fehlerhafter Code!

```
class BirneError implements EQ<Apfel>{
  String typ;

  BirneError(String typ){
    this.typ=typ;}

  public boolean eq(Birne that){
    return this.typ.equals(that.typ);
  }
}
```

Wir bekommen folgende Fehlermeldung:

```
sep@linux:~/fh/java1.5/examples/src> javac BirneError.java
BirneError.java:1: BirneError is not abstract and does not override abstract
method eq(Apfel) in EQ
class BirneError implements EQ<Apfel>{
^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

6.3.4 Generische Methoden

Bisher haben wir generische Typen für Klassen und Schnittstellen betrachtet. Generische Typen sind aber nicht an einen objektorientierten Kontext gebunden, sondern basieren ganz im Gegenteil auf dem Milner-Typsystem, das funktionale Sprachen, die nicht objektorientiert sind, benutzen. In Java verläßt man den objektorientierten Kontext in statischen Methoden. Statische Methoden sind nicht an ein Objekt gebunden. Auch statische Methoden lassen sich generisch in Java definieren. Hierzu ist vor der Methodensignatur in spitzen Klammern eine Liste der für die statische Methode benutzten Typvariablen anzugeben.

Eine sehr einfache statische generische Methode ist eine *trace*-Methode, die ein beliebiges Objekt erhält, dieses Objekt auf der Konsole ausgibt und als Ergebnis genau das erhaltene Objekt unverändert wieder zurückgibt. Diese Methode `trace` hat für alle Typen den gleichen Code und kann daher entsprechend generisch geschrieben werden:

Trace.java

```
class Trace {
  static <E>  E trace(E x){
    System.out.println(x);
    return x;
  }
}
```

```
jshell> Trace.trace((Trace.trace("hallo") +
   ...>    Trace.trace(" welt")).toUpperCase())
hallo
 welt
HALLO WELT
$2 ==> "HALLO WELT"
```

Wie man sieht, gibt es die Trace-Ausgabe und der gesamte Ausdruck hat einen Wert. Betrachten wir die Ergebnisvariable.

```
jshell> /var $2
|    String $2 = "HALLO WELT"
```

Diese ist vom Typ `String`.

Jetzt können wir die Methode aber ebensogut für einen Ausdruck der Typs `Integer` aufrufen.

```
jshell> Trace.trace(40+2)
42
$3 ==> 42
```

Die Ergebnisvariable ist entsprechend vom Typ `Integer`.

```
jshell> /var $3
|    Integer $3 = 42
```

In diesem Beispiel ist zu erkennen, dass der Typchecker eine kleine Typinferenz vornimmt. Bei der Anwendung der Methode `trace` ist nicht anzugeben, mit welchen Typ die Typvariable `E` zu instanziieren ist. Diese Information inferriert der Typchecker automatisch aus dem Typ des Arguments.

6.3.5 Beispiel einer eigenen Listenklasse

Die Paradeanwendung für generische Typen sind sogenannte Container- oder Sammlungsklassen, d.h. Klassen wie die erste generische Klasse `Box` die als Behälter für ein

oder mehrere Elemente eines variabel gehaltenen Typs dienen. Die Klassen `Box` und `Pair` sind in dieser Hinsicht recht langweilig, da sie nur ein bzw. zwei Elemente beinhalten. Eine häufig benötigte Funktionalität ist die Sammlung von potentiell beliebig vielen Elementen in einer bestimmten Reihenfolge. Eine solche Sammlung von Elementen wird als Liste bezeichnet. Mit Hilfe der generischen Typen können wir eine eigene generische Listenklasse implementieren.

Der Trick dabei ist, sich der Reihungen als internen Speicher zu bedienen. Reihungen sind ja in der Lage sind Elemente in einer festen Reihenfolge abzuspeichern. Das Problem der Reihungen ist aber, dass sie im Vorfeld beim Erzeugen nur für eine maximale Anzahl von Elementen erzeugt werden können. Eine Reihung kann dynamisch nicht nachträglich vergrößert werden.

In der Implementierung einer Listenklasse mit Hilfe einer Reihung geht man so vor, dass es eine interne Reihung gibt. Ist diese Reihung zu klein geworden, wird eine neue, größere Reihung erzeugt und alle Elemente in diese umkopiert.

zum Verständnis hilft vielleicht folgendes Bild. Sie haben ein Bücherregal. Dort ist Platz für eine bestimmte Anzahl von Büchern. Sie kaufen immer wieder neue Bücher und irgendwann ist Ihr Regal voll. Dann haben Sie folgende Option. Sie kaufen ein neues, größeres Regal. Nun bauen Sie das neue Regal auf und stellen Ihre Bücher in das neue Regal. Dort ist dann Platz für weitere Bücher. Das alte Regal entsorgen Sie, weil Sie nicht Platz für zwei Regale in der Wohnung haben. Genase gehen wir in unserer Listenimplementierung vor.

Unsere Listenklasse benötigt also zwei interne Felder:

- eines, das die Reihung[1] als Datenspeicher enthält,
- und eines, das notiert, wie viele Elemente in der Liste bereits enthalten sind.

OurList.java

```
package name.panitz.util;

public class OurList<A>{
  private Object[] store =  new Object[10];

  private int theSize = 0;
```

Die wichtigste Methode, die wir für eine Liste benötigen, soll ein Element in die Liste anfügen. Hierzu schreiben wir die Methode `add`, die ein Element am Ende der Liste hinzufügen soll.

[1] Leider passen aus historischen Gründen Reihungen und generische Typen in Java nicht ganz zusammen. Eine Reihung deren Elementtyp eine Typvariable ist, kann nicht mit `new` erzeugt werden. Daher werden wir hier für den internen Speicher den Typ `Object[]` verwenden.

OurList.java

```
public boolean add(A el){
```

Die Methode hat ein Problem, wenn die Reihung, die als Datenspeicher benutzt wird bereits komplett mit Listenelementen gefüllt ist. Dann brauchen wir mehr Platz. Hierzu wird eine interne Hilfmethode aufgerufen.

OurList.java

```
  if (theSize >= store.length){
    newStore();
  }
```

Wenn genügend Platz für ein neues Element ist, wird dieses am Ende der Liste in der Reiihung `store` abgelegt und dabei `theSize` um 1 erhöht.

OurList.java

```
  store[theSize++] = el;
  return true;
}
```

Die folgende Hilfsmethode legt einen neuen, größeren Speicher für die Elemente an und kopiert alle Elemente in diesen.

OurList.java

```
private void newStore() {
   Object[] newStore = new Object[store.length+10];

    for (int i= 0;i<store.length;i++) newStore[i] = store[i];

    store = newStore;
}
```

Jetzt sind die beiden Methoden, die wir als primäre Funktionalität für eine Liste erwartenm denkbar einfach zu implementieren. Die Anzahl der Elemente wird direkt in einem Feld gespeichert.

OurList.java

```
public int size() {
  return theSize;
}
```

Ein Element lässt sich direkt aus der Reihung heraus mit einem Index selektieren. Hier ist allerdings eine Typzusicherung nötig, da wir nur eine Reihung mit Objekttyp `Object` vorliegen haben.

OurList.java

```
  public A get(int i){
    return (A)store[i];
  }
}
```

Damit haben wir eine dynamisch wachsende Listenstruktur umgesetzt:

```
jshell> var xs = new name.panitz.util.OurList<String>()
xs ==> name.panitz.util.OurList@41cf53f9

jshell> xs.add("hallo");
   ...>     xs.add("Freunde");
   ...>     xs.add("Römer");
   ...>     xs.add("Landsleute");
   ...>     xs.add("leiht");
   ...>     xs.add("mir");
   ...>     xs.add("Euer");
   ...>     xs.add("Ohr");
   ...>     xs.add("Landsleute");
   ...>     xs.add("friends");
   ...>     xs.add("Romans");
$2 ==> true
...
$12 ==> true
```

Unser Listenobjekt kann jetzt seiner Elementenanzahl gefragt werden:

```
jshell> xs.size()
$13 ==> 11
```

Und es lässt sich ein Element an einem bestimmten Index selektieren.

```
jshell> xs.get(2)
$14 ==> "Römer"

jshell> for (int i = 0; i < xs.size(); i++) {
   ...>       System.out.print(xs.get(i).toUpperCase()+" ");
   ...>     }
HALLO FREUNDE RÖMER LANDSLEUTE LEIHT MIR EUER OHR LANDSLEUTE FRIENDS ROMANS
```

Weitere sinnvolle Methoden für die Klasse `OutList` sind Bestandteil von Übungsaufgaben.

Im nächsten Semester werden wir eine vollkommen andere Implementierung der Listenfunktionalität vorsehen. Dann werden Listen als rekursive Struktur umgesetzt.

6.3.6 Standard Sammlungsklassen

Java stellt im Paket `java.util` Implementierungen von Sammlungsklassen zur Verfügung. Dabei handelt ich sich um Klassen für Listen, Mengen und Abbildungen. Dieses sind Paradeanwendungen für generische Typen, denn es lassen sich über den Typparameter angeben, welchen Type die Elemente einer Sammlung im speziellen Fall haben.

Die Sammlungsklassen sind über verschiedene Schnittstellen definiert. Die Oberschnittstelle für Sammlungsklassen ist: `java.util.Collection`. Ihre Hauptunterschnittstellen sind: `List` und `Set` für die Darstellung von Listen bzw. Mengen.

Listen

Javas Standardlisten sind durch die Schnittstelle `jva.util.List` definiert. In dieser Schnittstelle finden sich eine Reihe von Methoden.

Zum einen Methoden, um Eigenschaften eines Listenobjekts zu erfragen, wie z.B.:

- `get(int index)` zum Erfragen eines Objektes an einem bestimmten Index der Liste.
- `size()`, um die Länge zu erfragen.
- `contains(Object o)`, um zu testen, ob ein bestimmtes Element in der Liste enthalten ist.

Desweiteren Methoden, die den Inhalt eines Listenobjektes verändern, wie z.B.:

- `add` zum Hinzufügen von Elementen.
- `clear` zum Löschen aller Elemente.
- `remove` zum Löschen einzelner Elemente.

Die eigentlichen konkreten Klassen, die Listen implementieren, sind:
`ArrayList`, `LinkedList` und `Vector`.

Dabei ist `ArrayList` die gebräuchlichste Implementierung. `Vector` ist eine ältere Implementierung, die als Nachteil hat, dass sie stets eine Synchronisation für nebenläufige Steuerfäden vornimmt, die in der Regel nicht unbedingt benötigt wird.

Die Klasse `ArrayList` entspricht der Klasse `OurList`, die im letzten Abschnitt entwickelt wurde. Es wird also intern eine Reihung benutzt, um die Elemente der Liste abzuspeichern. Wenn diese Reihung zu klein geworden ist, um alle Elemente der Liste hinzuzufügen, dann wird intern eine größere Reihung angelegt und die Elemente in diesen kopiert.

Die Klasse `LinkedList` hingegen realisiert die Liste auf eine gänzlich andere Art. Hier wird für jedes Element der Liste ein eigenes Listenkettenobjekt erzeugt. Diese Kettenglieder sind dann miteinander verbunden. Es entsteht eine rekursive Struktur. Dieses werden wir im zweiten Semester selbst programmieren.

Warum gibt es zwei verschiedene Listenklassen? Beide realisieren dieselbe Funktionalität. Jedoch haben sie beide sehr unterschiedliches Laufzeitverhalten:

- die `ArrayList` kann effizient auf beliebige Elemente in der Liste zugreifen. Die Methode `get(i)` hat einen konstanten Aufwand. Bei der `LinkedList` hingegen kann man über das Durchlaufen aller Elemente vom ersten Element an, an das i-te Element gelangen. Hier hat die Methode `get(i)` einen linearen Aufwand.
- `LinkedList` kann dafür im Vergleich effizienter Elemente hinzufügen oder löschen. Gerade beim Löschen eines Elements müssen in der `ArrayList` alle nachfolgenden Elemente in der internen Reihung umkopiert werden. Bei einer `LinkedList` kann ein Kettenglied einfach ausgehängt werden

Folgender kleiner Aufruf in der jshell zeigt, wie eine Liste erzeugt wird und ihr nach und nach Elemente hinzugefügt werden:

```
jshell> import java.util.*

jshell> List<String> xs = new ArrayList<>();
   ...>        //oder
   ...>       //  = new LinkedList<>();
   ...> xs.add("hallo");
   ...> xs.add("welt");
   ...> xs.add("wie");
   ...> xs.add("geht");
   ...> xs.add("es");
   ...> xs.add("dir");
xs ==> []
$18 ==> true
$19 ==> true
$20 ==> true
$21 ==> true
$22 ==> true
$23 ==> true

jshell> xs
xs ==> [hallo, welt, wie, geht, es, dir]
```

Wie man sieht, fügt die Methode `add` Objekte am Ende einer Liste an.

Mengen

Eine zweite Gruppen von Sammlungen stellen Mengen dar. Im Unterschied zu Listen gibt es keine doppelten Elemente in einer Menge und es gibt auch keine feste Reihenfolge der Elemente. Entsprechend des englischen Namens für Mengen heißt die Standardschnittstelle für Mengen in Java `Set`.

Auch hier gibt es zwei Implementierungen im Standard-API von Java. Die Klasse `HashSet` und die Klasse `TreeSet`. Die Unterschiede in der technischen Umsetzung werden Sie im Zuge des Moduls »Algorithmen und Datenstrukturen« kennenlernen.

Da es sich bei Mengen genauso um Sammlungen handelt, wie bei Listen, ist der praktische Umgang mit den entsprechenden Objekte der gleiche, wie schon bei Listen.

```
jshell> var xs = new TreeSet<String>()
xs ==> []
```

Nach Erzeugung einer Menge, können dieser Elemente hinzugefügt werden.

```
jshell> xs.add("hallo");
   ...> xs.add("Freunde");
   ...> xs.add("Römer");
   ...> xs.add("Landsleute");
   ...> xs.add("leiht");
   ...> xs.add("mir");
   ...> xs.add("Euer");
   ...> xs.add("Ohr");
   ...> xs.add("Landsleute");
   ...> xs.add("friends");
   ...> xs.add("Romans");
$26 ==> true
...
$34 ==> false
$35 ==> true
$36 ==> true
```

Das Ergebnis der Methode `add` zeigt an, ob ein Element tatsächlich in die Menge zugefügt wurde, oder nicht, weil ein gleiches Element bereits in der Liste enthalten war. Wie man sieht, enthält die Menge keine doppelten Elemente und entspricht die Reihenfolge nicht der EInfügereihenfolge.

```
jshell> xs
xs ==> [Euer,Freunde,Landsleute,Ohr,Romans,Römer,friends,hallo,leiht,mir]

jshell> xs.size()
$38 ==> 10
```

Noh einmal der Test: hinzufügen eines bereits enthaltenen Elements, ändert die Menge nicht.

```
jshell> xs.add("Ohr")
$39 ==> false

jshell> xs.size()
$40 ==> 10
```

Iterieren durch Sammlungen

Wenn man eine Sammlung von Objekten hat, sei es in Form einer Liste oder Menge, dann ist eine der häufigsten Aufgaben, durch allle Elemente dieser Sammlung einmal zu iterieren. Dabei kann man unterschiedlich vorgehen und nicht jedes Vorgehen ist dabei gleich gut. Im nächsten Semester werden wir uns noch detaillierter mit dem Thema der Iteration über Sammlungen beschäftigen.

Wir beginnen damit, eine Liste zu erzeugen:

```
jshell> import java.util.*

jshell> List<String> xs = new LinkedList<>();
   ...>     xs.add("hallo");
   ...>     xs.add("Freunde");
   ...>     xs.add("Römer");
   ...>     xs.add("Landsleute");
   ...>     xs.add("leiht");
   ...>     xs.add("mir");
   ...>     xs.add("Euer");
   ...>     xs.add("Ohr");
   ...>     xs.add("Landsleute");
   ...>     xs.add("friends");
   ...>     xs.add("Romans");
   ...>
xs ==> []
$9 ==> true
...
$19 ==> true
```

Das so erstellte Listenobjekt hat eine Methode für die Länge, d.h. die Anzahl der Elemente.

```
jshell> xs.size()
$20 ==> 11
```

Man kann auch Elemente an einem bestimmten Index erfragen.

```
jshell> xs.get(2)
$21 ==> "Römer"
```

```
jshell> for (int i = 0; i < xs.size(); i++) {
   ...>       System.out.print(xs.get(i).toUpperCase()+" ");
   ...>     }
HALLO FREUNDE RÖMER LANDSLEUTE LEIHT MIR EUER OHR LANDSLEUTE FRIENDS ROMANS
```

In der Klasse `Collections` gibt es eine statische Methode, mit der die Elemente einer Liste sortiert werden können.

```
jshell> Collections.sort(xs)
```

Eine Möglichkeit, durch eine Sammlungsklasse zu iterieren, ist die for-each-Schleife, die wir bisher nur für Reihungen kannte.

```
jshell>  for (String x : xs) {
   ...>       System.out.print(x+" ");
   ...>     }
Euer Freunde Landsleute Landsleute Ohr Romans Römer friends hallo leiht mir
```

Intern verwendet die for-each-Schleife ein Iteratorobjekt, dass es für jede Sammlungsklasse gibt. Wir können dieses Iteratorobjekt auch explizit verwenden, um über die Elemente einer Liste zu iterieren.

```
jshell> for (var it = xs.iterator();it.hasNext();  ){
   ...>       String x = it.next();
   ...>       System.out.print(x.toUpperCase()+" ");
   ...>     }
EUER FREUNDE LANDSLEUTE LANDSLEUTE OHR ROMANS RÖMER FRIENDS HALLO LEIHT MIR
```

Im nächsten Semester werden wir uns ausgiebiger mit Iteratoren beschäftigen.

Die Sammlungsklassen haben aber auch eine Methode `forEach`. Dieser kann in Form eines Lambda-Ausdrucks die Methode übergeben werden, die sagt, was mit den Elementen der Sammlung gemacht werden soll.

```
jshell> xs.forEach(x -> System.out.print(x.toUpperCase()+" "))
EUER FREUNDE LANDSLEUTE LANDSLEUTE OHR ROMANS RÖMER FRIENDS HALLO LEIHT MIR
```

Für alle Sammlungen lässt sich auch ein paralleler Strom erzeugen, auf dem es eine Methode `forEach` gibt. diese Methode wird dann eventuell sogar die als Lambda-Ausdruck mitgegebene Methode nebenläufig auf die Elemente der Sammlung ausführen.

```
jshell> xs.parallelStream().forEach(x->System.out.print(x.toUpperCase()+" "))
RÖMER FRIENDS ROMANS LEIHT MIR HALLO FREUNDE LANDSLEUTE LANDSLEUTE EUER OHR
```

Mit den Möglichkeiten der `Stream`-Objekte werden wir uns im nächsten Semester ausgiebig beschäftigen.

Abbildungen

Abbildungen assoziieren Elemente einer Art mit Elementen einer anderen Art. Sie sind vergleichbar mit Wörterbüchern. In einem deutsch/englischen Wörterbuch werden die deutschen Wörter auf die englischen Wörter mit der entsprechenden Bedeutung abgebildet.

Wir benutzen eine Abbildung, indem wir für einen Schlüssel (dem deutschen Wort) den entsprechenden Werteeintrag suchen (das englische Wort).

Abbildungen als Liste von Paaren Bevor wir uns den Standardklassen für Abbildungen widmen, geben wir eine eigene kleine Implementierung.

Da eine Abbildung jeweils ein Element mit einem anderem assoziiert, ist eine Umsetzung als Liste von Paaren naheliegend. Ein Paar besteht dabei aus einem Schlüssel und dem assoziierten Wert.

Paar.java

```
package name.panitz.util;
public class Paar<A,B>{
  public A fst;
  public B snd;
  public Paar(A f,B s){fst=f;snd=s;}
}
```

So ist eine einfache Umsetzung von Abbildungen durch Erweitern einer Standard-Listenklassen zu bekommen:

Abbildung.java

```
package name.panitz.util;
public class Abbildung<A,B> extends java.util.ArrayList<Paar<A,B>>{
  public void put(A schlüssel,B wert){
    add(new Paar<>(schlüssel,wert));
  }

  public B get(A schluessel){
    for (var p:this) {
      if (p.fst.equals(schluessel)){
        return p.snd;
      }
    }
    return null;
  }
}
```

Die Methode `lookup` iteriert über die Liste, bis sie ein Listenelement gefunden hat, dessen Schlüssel dem gesuchten Schlüssel gleich kommt.

```
jshell> var map =new name.panitz.util.Abbildung<String,String>()
map ==> []

jshell> map.put("Menge","set");
   ...> map.put("Abbildung","map");
   ...> map.put("Liste","list");
   ...> map.put("Iterator","iterator");
   ...> map.put("Schnittstelle","interface");
   ...> map.put("Klasse","class");
```

Jetzt können wir in dieser Abbildung wie in einem Wörterbuch nachschlagen.

```
jshell> map.get("Liste")
$8 ==> "list"

jshell> map.get("Aufzählung")
$9 ==> null
```

Schlüsseltyp und Wertetyp müssen nicht unbedingt wie im vorangestellten Beispiel gleich sein. Im folgenden Beispiel werden Schlüssel, die Strings sind auf Fließkommazahlen abgebildet und ergeben somit zum Beispiel eine Notenliste.

```
jshell> var noten = new name.panitz.util.Abbildung<String,Float>()
noten ==> []

jshell> noten.put("OOSE",1.0f)

jshell> noten.put("Analysis",1.7f)

jshell> noten.put("BWL",3.7f)

jshell> noten.get("OOSE")
$14 ==> 1.0

jshell> noten.get("ADS")
$15 ==> null
```

Standardklassen für Abbildungen Java stellt eine Schnittstelle zur Verwirklichung von Abbildungen zur Verfügung, die Schnittstelle `java.util.Map`.

Die wichtigsten zwei Methoden dieser Schnittstelle sind:

- `void put(K key, V value)`: ein neues Schlüssel-/Wertpaar wird der Abbildung hinzugefügt.
- `V get(K key)`: für einen bestimmten Schlüssel wird ein bestimmter Wert nachgeschlagen. Gibt es für diesen Schlüssel keinen Eintrag in der Abbildung, so ist das Ergebnis dieser Methode `null`.

Analog zu Mengen, gibt es im Standard-API zwei Implementierungen der Schnittstelle `Map`:
`HashMap` und `TreeMap`.[2]

Eine Schnittstelle, die Abbildungen implementiert, ist die Klasse `HashMap`. Ihre Benutzung funktioniert genauso wie die unserer Klasse `Abbildung`.

[2] Gedulden Sie sich bis zum Modul »Algorithmen und Datenstrukturen« im zweiten Semester, um den Unterschied zu verstehen.

```
jshell> var map = new java.util.HashMap<String,String>()
map ==> {}

jshell> map.put("Menge","set");
   ...> map.put("Abbildung","map");
   ...> map.put("Liste","list");
   ...> map.put("Iterator","iterator");
   ...> map.put("Schnittstelle","interface");
   ...> map.put("Klasse","class");
$19 ==> null
...
$24 ==> null

jshell> map.get("Liste")
$25 ==> "list"

jshell> map.get("Aufzählung")
$26 ==> null
```

Streams

6.4 Ein- und Ausgabe

Im Paket `java.io` befinden sich eine Reihe von Klassen, die Ein- und Ausgabe von Daten auf externen Datenquellen erlauben. In den häufigsten Fällen handelt es sich hierbei um Dateien.

Ein-/Ausgabeoperationen werden in Java auf sogenannten Datenströmen ausgeführt. Datenströme verbinden das Programm mit einer externen Datenquelle bzw.Senke. Auf diesen Strömen stehen Methoden zum Senden (Schreiben) von Daten an die Senke bzw. Empfangen (Lesen) von Daten aus der Quelle. Typischer Weise wird ein Datenstrom angelegt, geöffnet, dann werden darauf Daten gelesen und geschrieben und schließlich der Strom wieder geschlossen.

Ein Datenstrom charakterisiert sich dadurch, dass er streng sequentiell arbeitet. Daten werden also auf Strömen immer von vorne nach hinten gelesen und geschrieben.

Java unterscheidet zwei fundamentale unterschiedliche Arten von Datenströmen:

- **Zeichenenströme**: Die Grundeinheit der Datenkommunikation sind Buchstaben und andere Schriftzeichen. Hierbei kann es sich um beliebige Zeichen aus dem Unicode handeln, also Buchstaben so gut wie jeder bekannten Schrift, von lateinischer über kyrillische, griechische, arabische, chinesische bis hin zu exotischen Schriften wie der keltischen Keilschrift. Dabei ist entscheidend, in welcher Codierurung die Buchstaben in der Datenquelle vorliegen. Die Codierung wird in der Regel beim Konstruieren eines Datenstroms festgelegt. Geschieht dieses nicht, so wird die Standardcodierung des Systems, auf dem das Programm läuft, benutzt.

- **Byteströme (Oktettströme)**: Hierbei ist die Grundeinheit immer ein Byte, das als Zahl verstanden wird.

Vier abstrakte Klassen sind die Grundlagen der strombasierten Ein-/Ausgabe in Java. Die Klassen `Reader` und `Writer` zum Lesen bzw. Schreiben von Textdaten und die Klassen `InputStreamReader` und `OutputStreamReader` zum Lesen und Schreiben von Binärdaten. Folgende kleine Tabelle stellt dieses noch einmal dar.

	Eingabe	**Ausgabe**
binär	`InputStream`	`OutputStream`
Texte	`Reader`	`Writer`

Alle vier Klassen haben entsprechend ihrer Funktion Methoden `read` und `write`, die sich auf die enstprechenden Daten beziehen, sprich, ob `byte`-Werte gelesen und geschrieben werden oder `char`-Werte.

6.4.1 Dateibasierte Ein-/Ausgabe

Die vier obigen Grundklassen sind abstrakt, d.h. es können keine Objekte dieser Klassen mit einem Konstruktor erzeugt werden. Wir benötigen Unterklassen dieser vier Klassen, die die abstrakten Methoden implementieren. Eine der üblichsten Ein-/Ausgabe-Operation bezieht sich auf Dateien. Für alle für der Grundklassen gibt es im API eine Klasse, die die entsprechende Operation für Dateien realisiert. Diese Unterklassen haben im Namen einfach das Wort `File` voran gestellt.

Die einfachste Art der Eingabe dürfte somit das Lesen aus einer Textdatei sein. Hierzu eine einfache Methoden zum Lesen einer Textdatei:

```
import java.io.*;
package name.panitz.oose.io;
public class Textlesen {
```

Das Problem mit sämtlichen Ein-/Ausgabe-Operationen ist, dass die Kommunikation potentiell schief gehen kann. Die Datei kann nicht lesbar sein, gar nicht existieren, oder das externe Betriebsmittel, in diesem Falle die Festplatte, es könnte aber auch eine Netzwerkverbindung sein, nicht mehr antworten. Um koordiniert auf Fehlerfälle reagieren zu können, kennt Java das Konzept der Ausnahmen, die auftreten können und mit einer Fehlerbehandlung abgefangen werden können. Dieses Konzept werden wir erst im nächsten Kapitel kennenlernen. So lange wir noch nicht wissen, wie wir diese Ausnahmefälle behandeln, können wir jede Methode, in der Ein-/Ausgabe-Operationen stattfinden mit `throws Exception` markieren. Damit geben wir an, dass es bei Ausführung dieser Methode zu Fehlern kommen kann.

```
  public static String leseDatei(String dateiName)throws Exception{
    String result = "";
```

Wir wollen aus einer Datei lesen. Somit erzeugen wir ein Objekt der Klasse `FileReader` und speichern dieses als einen `Reader`.

```
    Reader read = new FileReader(dateiName);
```

Anders als man zunächst erwarten würde, hat die Klasse `Reader` keine Methode `read()`, die ein einzelnes `char` Zeichen zurück gibt, sondern eine Methode `read` mit dem Ergebnistyp `int`. Diese Zahl ist entweder die Unicode-Nummer des gelesenen Zeichens oder ein negativer Wert. Ist es ein negativer Wert, wird damit angezeigt, dass keine weiteres Zeichen mehr gelesen werden kann, dass wir am Ende des Zeichenstroms angelangt sind, weil in unserem Fall die Datei kein weiteres Zeichen mehr enthält. Deshalb sehen wir zunächst eine lokale Variable vom Typ `int` vor:

```
    int i;
```

Nun können wir nacheinander die Zeichen aus der Datei lesen. Mindestens einmal müssen wir lesen, um zu schauen, ob es überhaupt Zeichen in der Datei gibt. Dann lesen wir so lange, bis das Ergebnis des Lesens eine negative Zahl ist. Hierzu bietet sich die `do-while`-Schleife an, die mindestens einmal durchlaufen wird.

```
    do {
      i = read.read();
```

Wenn wir ein Zeichen gelesen haben und dieses Zeichen eine Unicode-Nummer ist, können wir diese Zahl als `char`-Wert interpretieren:

```
      if (i>=0){
        char c = (char)i;
        result = result+c;
      }
    } while (i>=0)
```

Wenn wir mit dem Lesen der Datei fertig sind, ist es sinnvoll, für das `Reader`-Objekt die Methode `close` aufzurufen, um dem Betriebssystem mitzuteilen, dass wir diese Ressource nicht weiter verwenden wollen.

```
    read.close();
```

Somit haben wir eine sehr simple Methode, die den kompletten Inhalt einer Datei einliest.

```
    return result;
  }
}
```

Ganz analog geht das Schreiben von Textdateien, so dass sich mit beiden zusammen recht einfach ein kleines Programm zum Kopieren von Textdateien schreiben lässt:

Copy.java

```
package name.panitz.oose.io;
import java.io.*;

public class Copy {
  public static void main(String[] args) throws Exception {
    Reader in = new FileReader(args[0]);
    Writer out = new FileWriter(args[1]);
    int c;
    while ((c = in.read()) >= 0){
      out.write(c);
    }
    out.close();
    in.close();
  }
}
```

6.4.2 Textcodierungen

`Reader` und `Writer` sind praktische Klassen zur Verarbeitung von Zeichenströmen. Primär sind aber auch Textdateien lediglich eine Folge von Bytes.

Mit den Klassen `InputStreamReader` und `OutputStreamWriter` lassen sich Objekte vom Typ `IntputStream` bzw. `OutputStream` zu `Reader`- bzw. `Writer`-Objekten machen.

Statt die vorgefertigte Klasse `FileWriter` zum Schreiben einer Textdatei zu benutzen, erzeugt die folgende Version zum Kopieren von Dateien einen über einen `FileOutputStream` erzeugten `Writer` bzw. einen über einen `FileInputStream` erzeugten `Reader`:

Copy2.java

```
import java.io.*;

class Copy2 {
  static public void main(String [] args) throws Exception {

    Reader reader = new InputStreamReader(new FileInputStream(args[0]));
    Writer writer  = new OutputStreamWriter(new FileOutputStream(args[1]));

    int c;
    while ((c = reader.read()) >= 0){
      writer.write((char)c);
    }
    writer.close();
    reader.close();
  }
}
```

Java-Strings sind Zeichenketten, die nicht auf eine Kultur mit einer bestimmten Schrift beschränkt, sondern in der Lage sind, alle im Unicode erfassten Zeichen darzustellen; seien es Zeichen der lateinischen, kyrillischen, arabischen, chinesischen oder sonst einer Schrift bis hin zur keltischen Keilschrift. Jedes Zeichen eines Strings kann potentiell eines dieser mehreren zigtausend Zeichen einer der vielen Schriften sein. In der Regel benutzt ein Dokument insbesondere im amerikanischen und europäischen Bereich nur wenige, kaum 100 unterschiedliche Zeichen. Auch ein arabisches Dokument wird mit weniger als 100 verschiedenen Zeichen auskommen.

Wenn ein Dokument im Computer auf der Festplatte gespeichert wird, so werden auf der Festplatte keine Zeichen einer Schrift, sondern Zahlen abgespeichert. Diese Zahlen sind traditionell Zahlen, die acht Bit im Speicher belegen, ein sogenanntes Byte. Ein Byte ist in der Lage, 256 unterschiedliche Zahlen darzustellen. Damit würde ein Byte ausreichen, alle Buchstaben eines normalen westlichen Dokuments in lateinischer Schrift (oder eines arabischen Dokuments) darzustellen. Für ein chinesisches Dokument reicht es nicht aus, die Zeichen durch ein Byte allein auszudrücken, denn es gibt mehr als 10000 verschiedene chinesische Zeichen. Es ist notwendig, zwei Byte im Speicher zu benutzen, um die vielen chinesischen Zeichen als Zahlen darzustellen.

Die *Zeichencodierung* (englisch: encoding) eines Dokuments gibt nun an, wie die Zahlen, die der Computer auf der Festplatte gespeichert hat, als Zeichen interpretiert werden sollen. Eine Codierung für arabische Texte wird den Zahlen von 0 bis 255 bestimmte arabische Buchstaben zuordnen, eine Codierung für deutsche Dokumente wird den Zahlen 0 bis 255 lateinische Buchstaben inklusive deutscher Umlaute und dem ß zuordnen. Für ein chinesisches Dokument wird eine *Codierung* benötigt, die den 65536 mit zwei Byte darstellbaren Zahlen jeweils chinesische Zeichen zuordnet. Man sieht, dass es *Codierungen* geben muss, die für ein Zeichen ein Byte im Speicher belegen, und solche, die zwei Byte im Speicher belegen. Es gibt darüberhinaus auch eine Reihe Mischformen; manche Zeichen

werden durch ein Byte, andere durch zwei oder sogar durch drei Byte dargestellt.

Die Klasse `OutputStreamWriter` sieht einen Konstruktor vor, dem man zusätzlich zum `OutputStream`, in den geschrieben werden soll, als zweites Element auch die Codierung angeben kann, in der die Buchstaben abgespeichert werden sollen. Wenn diese Codierung nicht explizit angegeben wird, so benutzt Java die standardmäßig auf dem Betriebssystem benutzte Codierung.

In dieser Version der Kopierung einer Textdatei wird für den `Writer` ein Objekt der Klasse `OutputStreamWriter` benutzt, in der als Zeichenkodierung utf-16 benutzt wird.

EncodedCopy.java

```
import java.nio.charset.Charset;
import java.io.*;

class EncodedCopy {
  static public void main(String [] args)throws Exception {
    Reader reader = new FileReader(args[0]);
    Writer writer = new OutputStreamWriter
                        (new FileOutputStream(args[1])
                        ,Charset.forName("UTF-16"));

    int c;
    while ((c = reader.read()) != -1){
      writer.write(c);
    }
    writer.close();
  }
}
```

Betrachtet man die Größe der geschriebenen Datei, so wird man feststellen, daß sie mehr als doppelt so groß ist wie die Ursprungsdatei.

```
:~/> java EncodedCopy EncodedCopy.java EncodedCopyUTF16.java
~/> ls -l EncodedCopy.java
-rw-r--r--    1 sep   users  443 2004-01-07 19:12 EncodedCopy.java
:~/> ls -l EncodedCopyUTF16.java
-rw-r--r--    1 sep   users  888 2004-01-07 19:13 EncodedCopyUTF16.java
```

Gängige Zeichencodierung sind:

- iso-8859-1: Damit lassen sich westeuropäische Texte mit den entsprechenden Sonderzeichen und Akzenten, wie in westeuropäischen Sprechen benötigt abspeichern. Jedes Zeichen wird mit einem Byte abgespeiechert. Andere Zeichen, sei es arabisch, chinesisch oder auch türkische Sonderzeichen, sind in dieser Zeichencodierung nicht abspeicherbar.
- utf-16: Hierbei hat jedes Zeichen genau zwei Byte in der Darstellung. Diese beiden Byte codieren exakt die Unicode-Nummer des Zeichens. Somit lassen sich in dieser

Zeichencodierung alle Unicode-Zeichen abspeichern. Allerdings, wenn man nur lateinische Schrift in einem Text hat, hat eines dieser beiden Bytes immer den Wert 0. Es wird also viel Platz verschwendet.

- utf-8: Hier werden gängigen lateinische Zeichen als ein Byte codiert. Alle anderen Sonderzeichen oder Zeichen aus anderen Schriften werden mit mehreren Bytes codiert. Dieses hat den Vorteil, dass man alle Zeichen kodieren kann, für Texte in lateinischer Schrift aber nur ein Byte pro Zeichen benötigt. Es hat den Nachteil, dass unterschiedliche Zeichen unterschiedlich viel Platz auf der Festplatte benötigen. Man kann also einer Datei nicht ansehen, das wievielte Byte zum Beispiel das 1000. Zeichen des Textes ist.

6.4.3 Gepufferte Ströme

Die bisher betrachteten Ströme arbeiten immer exakt zeichenweise, bzw. byteweise. Damit wird bei jedem `read` und bei jedem `write` direkt von der Quelle bzw. an die Senke ein Zeichen übertragen. Für Dateien heißt das, es wird über das Betriebssystem auf die Datei auf der Festplatte zugegriffen. Handelt es sich bei Quelle/Senke um eine teure und aufwändige Netzwerkverbindung, so ist für jedes einzelne Zeichen über diese Netzwerkverbindung zu kommunizieren. Da in der Regel nicht nur einzelne Zeichen über einen Strom übertragen werden sollen, ist es effizienter, wenn technisch gleich eine Menge von Zeichen übertragen wird. Um dieses zu bewerkstelligen, bietet Java an, Ströme in gepufferte Ströme umzuwandeln.

Ein gepufferter Strom hat einen internen Speicher. Bei einer Datenübertragung wird für schreibende Ströme erst eine Anzahl von Zeichen in diesem Zwischenspeicher abgelegt, bis dieser seine Kapazität erreicht hat, um dann alle Zeichen aus dem Zwischenspeicher *en bloc* zu übertragen. Für lesende Ströme wird entsprechend für ein `read` gleich eine ganze Anzahl von Zeichen von der Datenquelle geholt und im Zwischenspeicher abgelegt. Weitere `read`-Operationen holen dann die Zeichen nicht mehr direkt aus der Datenquelle, sondern aus dem Zwischenspeicher, bis dieser komplett ausgelesen wurde und von der Datenquelle wieder zu füllen ist.

Die entsprechenden Klassen, die Ströme in gepufferte Ströme verpacken, heißen:
`BufferedInputStream, BufferedOutputStream` und
entsprechend `BufferedReader, BufferedWriter`.

Jetzt ergänzen wir zur Effizienzsteigerung noch das Kopierprogramm, so daß der benutzte `Writer` gepuffert ist:

BufferedCopy.java

```
import java.io.*;
import java.nio.charset.Charset;

class BufferedCopy{
  static public void main(String [] args)throws Exception {
    Reader reader = new BufferedReader(new FileReader(args[0]));
    Writer writer = new BufferedWriter(new OutputStreamWriter
        (new FileOutputStream(args[1]),Charset.forName("UTF-16")));

    int c;
    while ((c = reader.read()) != -1){
      writer.write(c);
    }
    writer.close();
    reader.close();
  }
}
```

6.4.4 Lesen von einem Webserver

Bisher haben wir nur aus Dateien gelesen und damit den Vorteil der Abstraktion der Ströme, dass die Quelle aus der gelesen wird recht unterschiedlich sein kann, noch nicht demonstriert. In diesem Abschnitt zeigen wir, wie man statt aus einer Datei von einem Webserver Dokumente lesen kann. Auch dabei werden Ströme verwendet. Damit gibt es ein einheitliches API zum Lesen von Information aus ganz unterschiedlichen Quellen. Einstiegspunkt zum Lesen von einem Webserver ist die Klasse `java.net.URL`, mit der die Adresse des Webservers angegeben werden kann. Die Klasse hat einen Konstruktor, der ein String-Argument erhält.

```
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;
import java.net.URL;
import java.net.URLConnection;

public class ReadFromServer {
  public static void main(String[] args) throws Exception {

    URL url = new URL("http://www.jarass.com/home/index.php");
```

Die Klasse `URL` hat eine Methode `openConnection`, die es erlaubt eine Verbindung über das Netzwerk aufzubauen.

```
      URLConnection con = url.openConnection();
```

Von der damit erhaltenen Verbindung kann nun der Eingabestrom erfragt werden, also das Objekt, auf dem der Webserver uns Daten liefert.

```
      InputStream in = con.getInputStream();
```

Von nun an geht alles, wie bereits bei dem Einlesen aus einer Datei:

```
      Reader read = new InputStreamReader(in, "utf-8");
      int i;
      do {
        i = read.read();
        if (i>=0){
          char c = (char)i;
          System.out.print(c);
        }
      } while (i >= 0);
        read.close();
      }
    }
  }
```

6.4.5 Ströme für Objekte

Bisher haben wir uns darauf beschränkt, Zeichenketten über Ströme zu lesen und zu schreiben. Java bietet darüberhinaus die Möglichkeit an, beliebige Objekte über Ströme zu schreiben und zu lesen. Hierzu können mit den Klassen `ObjectOutputStream` und `ObjectInputStream` beliebige `OutputStream`- bzw. `InputStream`-Objekte zu Strömen für Objekte gemacht werden. In diesen Klassen stehen Methoden zum Lesen und Schreiben von Objekten zur Verfügung. Allerdings können über diese Ströme nur Objekte von Klassen geschickt werden, die die Schnittstelle `java.io.Serializable` implementieren. Die meisten Standardklassen implementieren diese Schnittstelle. `Serializable` enthält keine Methoden, es reicht also zum Implementieren aus, die Klausel `implements Serializable` für eine Klasse zu benutzen, damit Objekte der Klasse über Objektströme geschickt werden können.

Objektströme haben zusätzlich Methoden zum Lesen und Schreiben von primitiven Typen.

Folgendes Testprogramm schreibt eine Zahl und ein Listenobjekt in eine Datei, um diese anschließend wieder aus der Datei auszulesen.

WriteReadObject.java

```
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class WriteReadObject {
  public static void main(String [] args) throws Exception{
```

Zunächst erzeugen wir einen Ausgabestrom:

WriteReadObject.java

```
    OutputStream fos = new FileOutputStream("t.tmp");
```

Diesen verwenden wir nun, um eine in Strom, in den Objekte verschickt werden können, zu erzeugen.

WriteReadObject.java

```
    ObjectOutputStream oos = new ObjectOutputStream(fos);
```

Wir brauchen irgendein beliebiges Objekt, dass wir nun speichern können. Hierzu legen wir beispielsweise eine Liste an:

WriteReadObject.java

```
    List<String> xs = new ArrayList<String>();
    xs.add("the");
    xs.add("world");
    xs.add("is");
    xs.add("my");
    xs.add("oyster");
```

Und nun können wir sowohl eine Zahl, aber auch das ganze Listenobjekt in die Datei speichern:

WriteReadObject.java

```
    oos.writeInt(12345);
    oos.writeObject(xs);
    oos.close();
```

Nun können wir die geschriebene Datei wieder lesen und das Objekt dadurch neu erhalten. (realistischer Weise würde das natürlich in einem anderen Programm passieren). Hierzu brauchen wir einen EIngebaestrom:

WriteReadObject.java

```
      FileInputStream fis = new FileInputStream("t.tmp");
```

Diesen machen wir zu einem Strom, aus dem Objekte gelesen werden können:

WriteReadObject.java

```
    ObjectInputStream ois = new ObjectInputStream(fis);
```

Jetzt können wir daraus lesen. Zunächst die gespeicherte Zahl, dann das Listenobjekt.

WriteReadObject.java

```
    int i = ois.readInt();
    List<String> ys = (List<String>) ois.readObject();
    ois.close();

    System.out.println(i);
    System.out.println(ys);
  }
}
```

6.5 Ausnahmen

Es gibt während des Ablaufs eines Programms Situationen, die als Ausnahmen zum eigentlichen Programmablauf betrachtet werden können. Java hält ein Konzept bereit, das die Behandlung von Ausnahmen abseits der eigentlichen Programmlogik erlaubt.

6.5.1 Ausnahme- und Fehlerklassen

Java stellt Standardklassen zur Verfügung, deren Objekte einen bestimmten Ausnahme- oder Fehlerfall ausdrücken. Die gemeinsame Oberklasse aller Klassen, die Fehler- oder Ausnahmefälle ausdrücken, ist `java.lang.Throwable`. Diese Klasse hat zwei Unterklassen, nämlich:

- `java.lang.Error`: alle Objekte dieser Klasse drücken aus, dass ein ernsthafter Fehlerfall aufgetreten ist, der in der Regel von dem Programm selbst nicht zu beheben ist.
- `java.lang.Exception`: alle Objekte dieser Klasse stellen Ausnahmesituationen dar. Im Programm kann eventuell beschrieben sein, wie bei einer solchen Ausnahmesituation weiter zu verfahren ist. Eine Unterklasse von `Exception` ist die Klasse `java.lang.RuntimeException`.

6.5.2 Werfen von Ausnahmen

Ein Objekt vom Typ `Throwable` allein zeigt noch nicht an, dass ein Fehler aufgetreten ist. Hierzu gibt es einen speziellen Befehl, der im Programmablauf dieses kennzeichnet, der Befehl `throw`.

`throw` ist ein Schlüsselwort, dem ein Objekt des Typs `Throwable` folgt. Bei einem `throw`-Befehl verläßt Java die eigentliche Ausführungsreihenfolge des Programms und unterrichtet die virtuelle Maschine davon, dass eine Ausnahme aufgetreten ist. Z.B. können wir für die Fakultätsmethoden bei einem Aufruf mit einer negativen Zahl eine Ausnahme werfen:

FirstThrow.java

```
package name.panitz.exceptions;
public class FirstThrow {

  public static int fakultät(int n){
    if (n==0) return 1;
    if (n<0) throw new RuntimeException();
    return n*fakultät(n-1);
  }

  public static void main(String [] args){
    System.out.println(fakultät(5));
    System.out.println(fakultät(-3));
    System.out.println(fakultät(4));
  }
}
```

Wenn wir dieses Programm starten, dann sehen wir, dass zunächst die Fakultät für die Zahl 5 korrekt berechnet und ausgegeben wird, dann der Fehlerfall auftritt, was dazu führt, dass der Fehler auf der Kommandozeile ausgegeben wird und das Programm sofort beendet wird. Die Berechnung der Fakultät von 4 wird nicht mehr durchgeführt. Es kommt zu folgender Ausgabe:

```
swe10:~> java name.panitz.exceptions.FirstThrow
120
Exception in thread "main" java.lang.RuntimeException
  at name.panitz.exceptions.FirstThrow.fakultät(FirstThrow.java:6)
  at name.panitz.exceptions.FirstThrow.main(FirstThrow.java:12)
swe10:~>
```

Wie man sieht, unterrichtet uns Java in der ersten Zeile davon, dass eine Ausnahme des Typs `RuntimeException` geworfen wurde. In der zweiten Zeile erfahren wir, dass dieses bei der Ausführung der Methode `fakultät` in Zeile 6 der Klasse `FirstThrow` geschehen ist. Anschließend, in den Zeilen weiter unten, gibt Java jeweils an, in welcher Methode der Aufruf der in der drüber liegenden Methode stattfand.

Die Ausgabe gibt also an, durch welchen verschachtelten Methodenaufruf es an die Stelle kam, in der die Ausnahme geworfen wurde. Diese Aufrufstruktur wird als Aufrufkeller (*stack trace*) bezeichnet.

Das Erzeugen eines Ausnahmeobjekts allein bedeutet noch keinen Fehlerfall. Wenn wir das obige Programm minimal ändern, so dass wir das Schlüsselwort `throw` weglassen, so wird der Sonderfall für negative Eingaben nicht gesondert behandelt.

NonThrow.java

```
package name.panitz.exceptions;
public class NonThrow {

  public static int fakultät(int n){
    if (n==0) return 1;
    if (n<0) new RuntimeException();
    return n*fakultät(n-1);
  }

  public static void main(String [] args){
    System.out.println(fakultät(5));
    System.out.println(fakultät(-3));
    System.out.println(fakultät(4));
  }
}
```

Wenn wir dieses Programm starten, so wird es nicht terminieren und je nach benutzter Javamaschine schließlich abbrechen:

```
swe10:~> java name.panitz.exceptions.NonThrow
120

An irrecoverable stack overflow has occurred.
```

Es reicht also nicht aus, ein Fehlerobjekt zu erzeugen, sondern es muss dieses auch mit einem `throw`-Befehl geworfen werden. Geworfen werden können alle Objekte einer Unterklasse von `Throwable`. Versucht man hingegen, andere Objekte zu werfen, so führt dies schon zu einem Übersetzungsfehler.

Folgende Klasse:

Fehlerhafter Code!

```
package name.panitz.exceptions;
public class NotThrowable {

  public static void main(String [] args){
      throw "i am not throwable";
  }
}
```

führt zu einem Übersetzungsfehler:

```
swe10:~> javac -d . NotThrowable.java
NotThrowable.java:5: incompatible types
found   : java.lang.String
required: java.lang.Throwable
      throw "i am not throwable";
            ^
1 error
swe10:~>
```

Ausnahmen können natürlich nicht nur auftreten, wenn wir sie selbst explizit geworfen haben, sondern auch von Methoden aus Klassen, die wir selbst benutzen, geworfen werden. So kann z.B. die Benutzung der Methode `charAt` aus der Klasse `String` dazu führen, dass eine Ausnahme geworfen wird.

ThrowIndex.java

```
package name.panitz.exceptions;
public class ThrowIndex {

  public static void main(String [] args){
      "i am too short".charAt(120);
  }
}
```

Starten wir dieses Programm, so wird auch eine Ausnahme geworfen:

```
swe10:~> java name.panitz.exceptions.ThrowIndex
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
                                    String index out of range: 120
  at java.lang.String.charAt(String.java:516)
  at name.panitz.exceptions.ThrowIndex.main(ThrowIndex.java:5)
swe10:~>
```

Wie man an diesem Beispiel sieht, gibt Java nicht nur die Klasse der Ausnahme, die geworfen wurde, aus (`java.lang.StringIndexOutOfBoundsException:`), sondern auch noch eine zusätzliche Erklärung. Die Objekte der Unterklassen von `Throwable` haben in

der Regel einen Konstruktor, der erlaubt noch eine zusätzliche Information, die den Fehler erklärt, mit anzugeben. Das können wir auch in unserem Beispielprogramm nutzen:

SecondThrow.java

```
package name.panitz.exceptions;

public class SecondThrow {
  public static int fakultät(int n){
    if (n==0) return 1;
    if (n<0)
     throw
      new RuntimeException
           ("negative Zahl für Fakultätsberechnung");
    return n*fakultät(n-1);
  }

  public static void main(String [] args){
    System.out.println(fakultät(5));
    System.out.println(fakultät(-3));
    System.out.println(fakultät(4));
  }
}
```

Damit erhalten wir folgende Ausgabe:

```
swe10:~> java name.panitz.exceptions.SecondThrow
120
Exception in thread "main" java.lang.RuntimeException:
                                      negative Zahl für Fakultätsberechnung
  at name.panitz.exceptions.SecondThrow.fakultät(SecondThrow.java:6)
  at name.panitz.exceptions.SecondThrow.main(SecondThrow.java:12)
swe10:~>
```

7 Zusammenfassung und Ausblick

7.1 Fragen und Antworten

- **Ist Java eine kompilierte oder interpretierte Sprache?** Im Prinzip beides: Ein Compiler übersetzt den Quelltext in .class-Dateien, die Code für eine virtuelle Maschine enthalten. Ein Interpreter ist in der Lage diesen Code auszuführen.
- **Wie ist ein Javaprogramm strukturiert?** Pro Datei eine Klasse (oder Schnittstelle oder Aufzählung), die den Namen der Datei trägt.
- **Welche Namenskonventionen gibt es?** Klassennamen beginnen mit einem Großbuchstaben, globale Konstanten sind komplett in Großbuchstaben geschrieben. Pakete, Parameter, Variablen beginnen mit einem Kleinbuchstaben. Bei Bezeichnern aus mehreren Wörter fängt das nächste Wort immer mit einem Großbuchstaben an.
- **Ist Java statisch getypt?** Ja, der Compiler überprüft, ob das Programm korrekt getypt ist.
- **Gibt es auch dynamische Typüberprüfung zur Laufzeit?** Ja, zum einem verifiziert die virtuelle Maschine noch einmal den Code, zum anderen bewirkt eine Typzusicherung (cast) eine Laufzeittypüberprüfung.
- **Kann ich sicherstellen, daß die Typzusicherung während der Laufzeit nicht fehlschlägt.** Ja, indem vor der Typzusicherung der Test mit `instanceof` gemacht wird.
- **Wie schreibt man Unterprogramme oder Funktionen?** Immer innerhalb einer Klasse. Sie werden als Methoden bezeichnet.
- **Was sind die Eigenschaften einer Klasse?** Zum einen die Felder (auch als Attribute) bezeichnet, in denen Referenzen auf Objekte abgelegt werden können. Zum anderen die Methoden und Konstruktoren.
- **Was ist das `this`-Objekt?** Damit wird das Objekt bezeichnet, in dem sich eine Eigenschaft befindet.
- **Was bedeutet der `this`-Bezeichner, wenn ihm Parameter in Klammern folgen?** Dann bedeutet es den Aufruf eines weiteren Konstruktors aus derselben Klasse. Hierzu müssen Konstruktoren überladen sein. Dieser Aufruf kann nur als erster Befehl in einem Konstruktor stehen.

- **Was ist Vererbung?** Jede Klasse hat genau eine Oberklasse, die in der `extends`-Klausel angegeben wird. Objekte können alle Eigenschaften die in ihrer Oberklasse zur Verfügung stehen benutzen.
- **Was ist wenn ich keine Oberklasse angebe?** Dann ist automatisch die Klasse `Object` die Oberklasse.
- **Was sind Konstruktoren?** Konstruktoren sind der Code einer Klasse, der beim Erzeugen von neuen Objekten ausgeführt wird und in der Regel die Felder des Objektes initialisiert. Konstruktoren wir mit dem `new`-Befehl beim Erzeugen von Objekten aufgerufen.
- **Wie werden Konstruktoren definiert?** Ähnlich wie Methoden. Sie haben den Namen der Klasse und keinen Rückgabetyp.
- **Wie werden Konstuktoren aufgerufen?** Durch das Schlüsselwort `new` gefolgt von dem Klassennamen.
- **Hat jede Klasse einen Konstruktor?** Ja.
- **Und wenn ich keinen Konstruktor für meine Klasse schreibe?** Dann fügt Java einen leeren Konstruktor ohne Parameter ein.
- **Was ist der Unterschied zwischen statischen und nicht statischen Eigenschaften einer Klasse?** Eine statische Eigenschaft ist nicht an spezielle Objekte gebunden. Sie hat daher auch kein `this`-Objekt. Eine statische Eigenschaft existiert nur einmal für alle Objekte einer Klasse. Nicht-statische Methoden werden auch als Objektmethoden bezeichnet.
- **Gibt es auch statische Konstruktoren?** Ja, pro Klasse genau einen, der keine Parameter hat.
- **Kann ich den statischen Konstruktor auch selbst definieren?** Ja, mit dem Schlüssenwort `static` gefolgt von in geschweiften Klammern eingeschlossenen Code.
- **Was bedeutet Überschreiben von Methoden?** Methoden, die es in der Oberklasse bereits gibt in einer Unterklasse neu zu definieren.
- **Kann ich auch Konstruktoren überschreiben?** Nein.
- **Was bezeichnet man als Polymorphie?** Wenn eine Methode in verschiedenen Unterklassen einer Klasse überschrieben wird.
- **Kann ich in einer überschreibenen Methode, die überschriebene Methode aufrufen?** Ja, indem man das Schlüsselwort `super` benutzt und mit einem Punkt abgetrennt den eigentlichen Methodenaufruf folgen lässt.
- **Kann ich Konstruktoren der Oberklasse aufrufen.** Ja, aber nur im Konstruktor als erste Anweisung. Hier muss sogar der Aufruf eines Konstruktors der Oberklasse stehen. Dieser Aufruf wird durch das Schlüsselwort `super` gefolgt von der Parameterliste gemacht.

- **Was ist, wenn ich im Konstruktor keinen Aufruf an einen Konstruktor der Oberklasse schreiben.** Dann generiert Java den Aufruf eines Konstruktors der Oberklasse ohne Parameter als erste Anweisung in den Konstruktor. Sollte so ein parameterloser Konstruktor nicht existieren, dann gibt es allerdings einen Folgefehler.
- **Was ist späte Bindung (late binding)?** Beim Aufruf von Objektmethoden wird immer der Methodencode ausgeführt, der in der Klasse implementiert wurde, von der das Objekt, auf dem diese Methode aufgerufen wurde, erzeugt wurde. Es wird also immer die überschreibende Version einer Methode benutzt.
- **Funktioniert späte Bindung auch für Felder?** Nein.
- **Funktioniert späte Bindung auch für statische Methoden?** Nein.
- **Funktioniert späte Bindung auch für Konstruktoren?** Nein.
- **Was sind überladene Methoden?** Methoden gleichen Namens in einer Klasse, die sich in Typ/Anzahl der Parameter unterscheiden.
- **Können auch Konstruktoren überladen werden?** Ja.
- **Gibt es das Prinzip von später Bindung auch für die verschiedenen überladenen Versionen einer Methode?** Nein! Die Auflösung, welche der überladenen Versionen einer Methode ausgeführt wird, wird bereits statisch vom Compiler vorgenommen und nicht dynamisch während der Laufzeit.
- **Was sind abstrakte Klassen?** Klassen, die als `abstract` deklariert sind. Nur abstrakte Klassen können abstrakte Eigenschaften enthalten.
- **Und was sind abstrakte Eigenschaften?** Das sind Methoden, die keinen Methodenrumpf haben.
- **Können abstrakte Klassen Konstruktoren haben?** Ja, allerdings können von abstrakten Klassen keine Objekte mit `new` erzeugt werden.
- **Wie kann ich dann Objekte einer abstrakten Klassen erzeugen und wozu haben die dann Konstruktoren?** Indem Objekte einer nicht abstrakten Unterklasse mit `new` erzeugt werden. Im Konstruktor der Unterklasse wird ein Konstruktor der abstrakten Oberklasse aufgerufen.
- **Kann eine Klasse mehrere abstrakte Oberklassen haben?** Nein, auch abstrakte Klassen sind Klassen und es gilt die Regel: jede Klasse hat genau eine Oberklasse.
- **Kann ich in einer abstrakten Klasse abstrakte Methoden aufrufen?** Ja, im Rumpf einer nicht-abstrakten Methode können bereits abstrakte Klassen aufgerufen werden.
- **Was sind Schnittstellen?** Man kann Schnittstellen als abstrakte Klassen ansehen, in denen jeder Methode abstrakt ist.

- **Warum gibt es dann Schnittstellen?** Schnittstellen gelten nicht als Klassen. Eine Klasse kann nur eine Oberklasse haben, aber zusätzlich mehrere Schnittstellen implementieren.
- **Wie wird deklariert, daß eine Klasse eine Schnitstelle implementiert?** Durch die `implements`-Klausel in der Klassendeklaration. In ihr können mehrere Komma getrennte Schnittstellen angegeben werden.
- **Kann eine Schnittstelle eine Oberklasse haben?** Nein.
- **Kann eine Schnittstelle weitere Oberschnittstellen haben.** Ja, diese werden in der `extends`-Klausel angegeben.
- **Muss eine Klasse die eine Schnittstelle implementiert alle Methoden der Schnittstelle implementieren?** Im Prinzip ja, jedoch nicht, wenn die Klasse selbst abstrakt ist.
- **Haben Schnittstellen auch Objektfelder?** Nein! Nur Methoden und statische Felder.
- **Sind alle Daten in Java Objekte?** Nein, es gibt 8 primitive Typen, deren Daten keine Objekte sind. Es gibt aber zu jeden dieser 8 primitiven Typen eine Klasse, die die Daten entsprechend als Objekt speichern.
- **Welches sind die primitiven Typen?** byte, short, int, long, double, float, char, boolean
- **Sind Strings Objekte?** Ja, die Klasse String ist eine Klasse wie Du und ich.
- **Sind Reihungen (arrays) Objekte?** Ja, und sie haben sogar ein Attribut, das ihre Länge speichert.
- **Was sind generische Typen?** Klassen oder Schnittstellen, in denen ein oder mehrere Typen variabel gehalten sind.
- **Wie erzeugt man Objekte einer generischen Klasse?** Indem beim Konstruktoraufruf in spitzen Klammern konkrete Typen für die Typvariablen angegeben werden.
- **Was passiert, wenn ich für generische Typen die spitzen Klammern bei der Benutzung weglasse?** Dann gibt der Compiler eine Warnung und nimmt den allgemeinsten Typ für die Typvariablen an. Meistens ist das der Typ `Object`.
- **Was sind generische Methoden?** Methoden in denen ein oder mehrere Parametertypen variabel gehalten sind.
- **Was sind typische Beispiel für generische Typen?** Alle Sammlungsklassen und Abbildungsklassen im Paket `java.util`, zB die Klassen `ArrayList`,`LinkedList`, `HashSet`, `Vector` oder die Schnittstellen `List`, `Map`, `Set`.
- **Wo wir gerade dabei sind. Was sollte ich bei der Klasse `Vector` beachten?** In 90% der Fälle ist ein Objekt der Klasse `ArrayList` einem Objekt der Klasse `Vector` vorzuziehen. `Vector` ist eine sehr alte Klasse. Ihre Objekte sind synchronisiert, die anderen Sammlungsklassen nicht, es lassen sich von den anderen

Sammlungsklassen allerdings synchronisierte Kopien machen. Am besten `Vector` gar nie benutzen.

- **Was ist automatisches Boxing und Unboxing?** Die Konvertierung von Daten primitiver Typen in Objekte der korrespondierenden Klassen und umgekehrt wird automatisch vorgenommen.
- **Welche zusammengesetzten Befehle gibt es?** if, while, for, switch
- **Was hat es mit der besonderen for-Schleife auf sich?** Es handelt sich um eine sogenannte for-each-Schleife. Syntaktisch trennt hier ein Doppelpunkt die lokale Schleifenvariable das Sammlungsobjekt.
- **Für welche Objekte kann die for-each-Schleife benutzt werden?** Für alle Objekte, die die Schnittstelle `Iterable` implementieren und für Reihungen.
- **Bedeutet das, ich kann die for-each Schleife auch für meine Klassen benutzen?** Ja, genau, man muß nur die Schnittstelle `Iterable` hierzu implementieren.
- **Können Operatoren überladen oder neu definiert werden?** Nein.
- **Was sind Ausnahme?** Objekte von Unterklassen der Klasse `Exception`. Zusätzlich gibt es die Klasse `Error` und die gemeinsame Klasse `Throwable`.
- **Wozu sind Ausnahmen gut?** Um in bestimmten Situationen mit einem `throw` Befehl den Programmfluß abzubrechen um eine besondere Ausnahmesituation zu signalisieren.
- **Was für Objekte dürfen in eine `throw` geworfen werden?** Nur Objekte einer Unterklasse von `Throwable`.
- **Wie werden geworfene Ausnahme behandelt?** Indem man sie innerhalb eines `try-catch`-Konstruktes wieder abfängt.
- **Kann man auf unterschiedliche Ausnahmen in einem `catch` unterschiedlich reagieren?** Ja, einfach mehrere `catch` untereinanderschreiben. Das als erstes zutreffende `catch` ist dann aktiv.
- **Was ist das mit dem `finally` beim `try-catch`?** Hier kann Code angegeben werden, der immer ausgeführt werden soll, egal ob und welche Ausnahme aufgetreten ist. Dieses ist sinnvoll um eventuell externe Verbindungen und ähnliches immer sauber zu schließen.
- **Darf man beliebig Ausnahmen werfen?** Ausnahmen, die nicht in einem `catch` abgefangen werden müssen in der `throws`-Klausel einer Methode deklariert werden.
- **Alle?** Nein, Ausnahmeobjekte von `RuntimeException` dürfen auch geworfen werden, wenn sie nicht in der `throws`-Klausel einer Methode stehen.
- **Wozu sind Pakete da?** Unter anderen damit man nicht Milliarden von Klassen in einem Ordner hat und dabei die Übersicht verliert.

- **Ist mir egal. Pakete sind mir zu umständlich.** Das ist aber dumm, wenn man Klassen gleichen Namens aus verschiedenen Bibliotheken benutzen will. Die kann man nur unterscheiden, wenn sie in verschiedenen Paketen sind.
- **OK, ich sehs ein. Wie nenne ich mein Paket?** Am besten die eigene Webdomain rückwärsts nehmen also für uns an der FH:
  `de.hsrm.informatik.panitz.meineApplikation`
- **Da fehlt doch der Bindestrich.** Der ist kein gültiges Zeichen innerhalb eines Bezeichners in Java (es ist ja der Minusoperator). Bindestriche in Webadressen sind eine recht deutsche Krankheit.
- **Muss ich imports machen um Klassen aus anderen Paketen zu benutzen?** Nein. Das macht die Sache nur bequemer, weil sonst der Klassenname immer und überall komplett mit seinem Paket angegeben werden muß, auch beim Konstruktoraufruf oder z.B. `instanceof`.
- **Machen Import-Anweisungen das Programm langsamer oder größer?** Nein! Sie haben insbesondere nichts mit includes in C gemein. Sie entsprechen eher dem `using namespace` aus C++.
- **Da gibt es doch auch noch `public`, `private` und `protected`.** Jaja, die Sichtbarkeiten. Hinzu kommt, wenn man keine Sichtbarkeit hinschreibt. `public` heißt von überall aufrufbar, `protected` heißt in Unterklassen und gleichem Paket aufrufbar, `package` das ist wenn man nichts hinschreibt, heißt nur im gleichen Paket sichtbar, und `private` nur in der Klasse.
- **Und dann war da noch das `final`?** Das hat zwei Bedeutungen: bei Variablen und Feldern, daß sie nur einmal einen Wert zugewiesen bekommen, bei Klassen, daß keine Unterklassen von der Klasse definiert werden
- **Ich will GUIs programmieren.** Wunderbar, da gibt es eine ganze Reihe Bibliotheken, unter anderen auch Swing. JavaFX und AWT.
- **Na toll, warum drei?** Historisch.
- **Und welche sol ich jetzt benutzen?** Swing! Nein eigentlich JavaFX.
- **Also kann ich alles über AWT vergessen, insbesondere das Paket `java.awt`?** Nein. Swing benutzt große Teile aus AWT. Insbesondere hat jede Swingkomponente eine Oberklasse aus AWT. Aber auch die Ereignisbehandlungsklassen werden vollständig aus AWT benutzt.
- **Und woran erkenne ich jetzt, wann ich Komponente aus AWT benutzen muss, obwohl ich in Swing programmiere?** Alle graphischen AWT-Komponenten haben ein Swing Pendant, das mit dem dem Buchstaben `J` beginnt, z.B. `javax.swing.JButton` im Gegensatz zu `java.awt.Button`.
- **Sonst noch paar Tipps, zum benutzen von Swing?** Keine eigenen Threads schreiben! Nicht die Methode `paint` sondern `paintComponent` überschreiben.

- **Toll, wenn ich keine Threads schreiben soll, wie mache ich dann z.B. zeitgesteuerte Ereignisse in Swing?** Da gibt es die Klasse `javax.swing.Timer` für.
- **Na ich weiß nicht, ob ich jetzt genügend weiß für die Klausur?** Das weiß ich auch nicht, über alles haben wir hier nicht gesprochen. Ich denke aber es reicht an Wissen aus.