Java Programmiergrundlagen

Einführung

Nachdem wir im vorhergehenden Workshop (Java-Einführung) die ersten praktischen Erfahrungen mit Java gemacht haben, wollen wir uns jetzt ein bisschen mit der Theorie zu Java befassen.

Datentypen

Bevor man mit irgendetwas anderem anfängt, sollte man sich um die Datentypen in Java bekümmern, schließlich steht die Definierung der Variablen meist am Anfang jedes Quellcodes. Da, wie du schon wissen solltest, Java eine sehr auf Sicherheit getrimmte Sprache ist, nimmt es Java auch mit dem Datentypen recht ernst.
Es macht jetzt jedoch an dieser Stelle - meines Erachtens - keinen Sinn alle Datentypen inklusive ihres Wertebereichs und des Platzbedarfs auf zu listen; man braucht für den Hausgebrauch nur die wenigsten Datentypen, weshalb ich auch nur die wichtigsten behandeln werde.
Fangen wir erst einmal mit Zahlen an: Da gibt es zum einen *short* für die "kleinen Zahlen" (bis ± 32.768), wenn die Zahlen etwas größer sein müssen, dann gibt es *int* (bis ± 2.147.483.648). Wenn auch das noch nicht ausreichen sollte bietet sich *long* an (für wirklich große Zahlen bis ± 92.233.772.036.854.775.807). Der Speicherbedarf verdoppelt sich dabei immer (von 8 auf 16 auf 32 und schließlich auf 64 Bit). Bis jetzt waren unsere Datentypen nur für Ganzzahlen geeignet. Für Fließkommazahlen (Kommazahlen) gibt es *float* und *double*. *float* benötigt 32 und *double* 64 Bit.
Datentypen gibt es selbstverständlich nicht nur für Zahlen, sondern auch für Buchstaben. Wenn in der Variable nur ein Buchstabe gespeichert werden soll, dann bietet sich char an. Dort kann ein Zeichen für 16 Bit gespeichert werden. Jeder der sich auch nur ein wenig mit Programmierung auskennt wird noch eine Datentypenart vermissen: Zeichenketten-Datentypen in denen Wörter und Sätze (jedenfalls mehr als ein Zeichen) gespeichert werden können. Diesen Datentyp wird man in Java vergeblich suchen. Doch keine Angst, man kann natürlich auch Zeichenketten speichern. Für diese Zwecke gibt es *String*, das ist jedoch kein gewöhnlicher Datentyp, sondern eine Klasse mit Methoden. Das heißt, die Entwickler von Java waren so freundlich uns mehr als einen Datentypen zur Verfügung zu stellen, sie haben uns gleich ein Objekt geliefert, mit dem man den Speicherinhalt auch gleich weiterverarbeiten kann. Ganz schön clever, schließlich wird dieses Objekt auch am häufigsten verwendet.

Variablen

Jetzt kennen wir die wichtigsten Datentypen in Java, diese wollen wir nun nützlich einsetzen. Das geht natürlich am besten im Zusammenhang mit Variablen. Für alle die mit dem Begriff Variable noch nichts anfangen können ganz kurz: Eine Variable ist der Name für eine Speicherstelle im Hauptspeicher; sie macht es dem Programmierer einfach auf den Inhalt zuzugreifen in dem einfach nur der Variablenname aufgerufen werden muss. Der Inhalt der Speicherstelle kann sich ändern, der Variablenname bleibt immer gleich.
Bevor man die Variablen benutzen kann muss man sie definieren, damit Java schon einmal für die Variable die benötigte Speicherstelle zu reservieren, die Länge richtet sich nach dem Platzbedarf des Datentypen. Aus diesem Grund steht man vor dem Problem, ob man einen Datentyp verwenden soll, der einen möglichst großen Wertebereich hat, oder den Datentyp der am wenigsten Speicher verbraucht.
Variablen werden allgemein so definiert: erst den *Datentyp*, dann den *Variablenna-*

men (also z.B. *int a;* usw.).
Da muss man jetzt natürlich mit String aufpassen! Wer *sehr* aufmerksam den vorherigen Abschnitt gelesen hat, der hat gemerkt, dass ich alle Datentypen kleingeschrieben habe (so müssen sie dann auch in der Definition geschrieben werden), bis auf String. Das habe ich groß geschrieben, was kein Tippfehler ist, sondern Absicht.
Wir erinnern uns: String ist gar kein gewöhnlicher Datentyp, sondern eine Klasse mit Methoden! Aus diesem Grund wird String groß geschrieben (also ***S****tring b*;).
Ich möchte an dieser Stelle noch etwas zur Wahl des Bezeichners (Variablennamens) loswerden: 1. dieser unterliegt Konventionen, so muss er mit einem Buchstaben beginnen und darüber hinaus nur aus Buchstaben, Ziffern und dem Unterstrich (_) enthalten. 2. Java beachtet die Groß- und Kleinschreibung ganz genau: name, Name oder NAME sind unterschiedliche Bezeichner! Dadurch entstehen Fehler (auch das Semikolon nicht vergessen).
Noch ein weiterer Trick für die Definition: Werden mehrere Variablen mit dem gleichen Datentypen benötigt, kann man sie auf einmal definieren und zwar z.B. so *int a, b;* (also die Aufzählung durch Kommata getrennt).
Bis jetzt haben wir die Variablen nur definiert, doch einfach so bringen sie mir garnichts. Man muss den Variablen etwas zuweisen und das geht so: a = 7; bei Zahlen oder b = 'Wort'; für Zeichenketten. Zeichenketten werden also in Hochkommata (lässt sich mit [Shift] + [#] machen) eingeschlossen. Achtung bei Fließkommazahlen: **Kein** Dezimal-Komma sondern ein Dezimal-**Punkt**. Ansonsten ist mit "unerklärlichen" Fehlermeldungen zu rechnen.
Wie in der Einführung zu Java schon erwähnt, hat Java seine Ursprünge in der Programmiersprache C++. Von dieser hat er auch folgende Form der Zuweisung geerbt:
intZahl += 5
Das ist die Abkürzung für: intZahl = intZahl + 5. So lässt sich einige Arbeit bei der Zuweisung sparen.

Allgemeine Hinweise

So bevor ich nun mit den komplexeren Programmiergrundlagen fortfahre, möchte ich noch ein paar allgemeine Hinweise zur Programmierung mit Java loswerden.
Wenn man Fehler leichter finden will, Übersicht in sein Programm bringen will oder nicht alleine an einem Projekt arbeitet, dann empfiehlt es sich den Quelltext einzurücken. Bei if-Bedingungsabfragen, Methoden und Klassen bietet sich das Einrücken zwingend an. JCreator macht es bei diesen Bestandteilen automatisch (wenn man diesen Editor denn verwendet).
Wenn eine Anweisung, z.B. eine if-Bedingungsabfrage oder eine Methode, mehrere Zeilen Quelltext umfasst sind diese in geschweifte Klammern ({ })zu schreiben. Das sieht dann z.B. so aus:

```
if (Bedingung)
{
 Anweisung1
 Anweisung2
}
```

Ich kann garnicht oft genug betonen, wie wichtig in Java die korrekte Schreibweise ist! Variablen oder Schlüsselwörter zum Beispiel, müssen immer exakt - das beinhaltet auch die Groß- und Kleinschreibung - so wie sie vereinbart sind geschrieben wer-

lokalisierenden Fehlern führt.

Variablenumwandlung

Nach ein paar Tipps zur Auflockung nun weiter im Programm. Nicht immer ist der Datentyp für die Situation passend: Wenn man mit einer Benutzereingabe (meist im String-Format) rechnen möchte (geht nur mit numerischen Datentypen) steht man vor einem Problem. Man muss die Eingabe irgendwie in das numerische Format umwandeln. Wer mal mit Visual Basic gearbeitet hat, der wird sich an den "tollen" Datentyp Variant erinnern, der alles automatisch umwandelt (dabei aber dummerweise Speicherplatz verschwendet). Dies gibt es in Java NICHT!
In Java muss der Datentyp in den man umwandeln will bekannt sein:
numInt = Integer.parseInt (strEing);

Hier wird eine Benutzereingabe *strEing* in einen numerischen Wert umgewandelt, der der Variable *numInt* zugewiesen wird. Die Umwandlung wird mit dem Methodenteil *parse[...]* (bei uns *parseInt*) durchgeführt, der zum Objekt des Datentyps gehört.
Konkret heißt das: *numInt* ist eine *Integer*-Variable, also ist das Datentypobjekt *Integer.*, dann der Methodenteil *parse* und für Integer *Int* und in Klammern die Variable die umgewandelt werden soll.
Diese Umwandlung funktioniert nicht nur ins Integer-Format, sondern auch für alle anderen einfachen Datentypen (byte, short, int long, float und double.
Will man ASCII-Zeichen auf dem Bildschirm ausgeben, so muss man den ASCII-Wert (also die Zahl aus der ASCII-Tabelle) in ein ASCII-Zeichen (also a,b oder c) umwandeln. Dieser Vorgnag wird Typecast genannt und sieht so aus: (char)zeichen

Kontrollstrukturen

Bis jetzt wurde davon ausgegangen, dass ein Programm einfach nacheinander ausgeführt wird. Das ist jedoch nicht immer sinnvoll. Oft ist es notwendig, dass das Programm auf Situationen reagiert. Wenn der Benutzer folgendes tut, dann soll das Programm folgendes tun (vielleicht auch mehrfach).
Genau dafür wurden Kontrollstrukturen geschaffen. Diese kann man in zwei verschiedene Techniken unterteilen. Zum einen die Auswahl, d.h. wenn folgendes eintritt, dann tue dies (oder das). Zum anderen die Wiederholung, d.h. solange die Situation besteht (oder nicht besteht) tue dies.

Auswahl

Wie ich im obigen Abschnitt erwähnt habe, werden mit Hilfe der Auswahl in Abhängigkeit von einer Bedingung Anweisungen ausgeführt oder weggelassen. Das Programm kann also selbstständig auf Bedingungen reagieren.
Solche Bedingungen bestehen meist aus Vergleichen mit den Operatoren >, <, >= (größer gleich), <= (kleiner gleich), == (gleich) und *!*= (ungleich). Also z.B. Ist Uhrzeit == 12 dann rufe zu Mittag.
Bei dem Gleichoperator ist darauf zu achten, dass man zwei Gleichheitszeichen hintereinander schreibt; manche andere Programmiersprache erlaubt den Vergleich mit Gleichheitszeichen, Java ist da aber sehr genau und reserviert das einzeln stehende Gleichheitszeichen lediglich für die Zuweisung.

Auswahl mit if

In Java wird eine Auswahl mit if so geschrieben:
if (Bedingung) Anweisung;

Wie ich oben bei den Tipps schon vorweg genommen habe, werden mehrere Anweisungen in geschweifte Klammern eingeschlossen.
Wir haben hier die allgemeine Anweisung für eine sogenannte einseitige Auswahlstruktur, d.h. es wird nur dann etwas getan, wenn die Bedingung stimmt.
Im Gegensatz dazu wird bei der sogenannten zweiseitigen Auswahlstruktur auch dann noch was getan, wenn die Bedingung nicht stimmt. Also z.B. beim Geldautomaten: Ist Kontenstand > 0 dann Abheben erlaubt, sonst Meldung über zu niedrigen Kontenstand ausgeben. Die zweiseitige Auswahlstruktur sieht in Java so aus:
if (Bedingung) Anweisung1; else Anweisung2;

Wie man sieht, wird der sonst-Fall mit else eingeleitet. Ich möchte noch auf gerne gemachte Fehler hinweisen:
Hinter if (Bedingung) steht nie ein Semikolon! Wenn man also mehrzeilige Anweisungsblöcke hat, dann kommen nur hinter die Anweisungszeilen Semikolons!
Anweisungsblöcke werden nur dann als Block ausgeführt, wenn man diese in geschweifte Klammern schreibt. Ansonsten wird nur die erste Zeile des Anweisungsblocks von der Bedingung abhängig gemacht und die anderen werden immer ausgeführt!

Bedingungsverknüpfung

An dieser Stelle möchte ich nun noch was zu der Bedingung sagen: Bisher haben wir nur eine Bedingung gehabt. Manchmal sind aber mehrere Bedingungen zu erfüllen, bevor die Auswahl stattfinden soll. Wenn wir wieder auf unser vorletztes Beispiel zurückkommen: Ist (Uhrzeit == 12 && Wochentag == Sontag) dann rufe zum Sonntagsbraten.
Hier siehst du auch schon die richtigen boolschen Operatoren für die UND-Verknüpfung. Darüber hinaus gibt es noch den ODER-Operator || ([Alt Gr] + [<]). Der Unterschied ist, dass beim UND-Operator beide Bedingungen gleichzeitig erfüllt sein müssen. Beim ODER-Operator reicht es wenn nur eine der Bedingungen erfüllt ist.

Mehrseitige Auswahl

Es gibt zwei Formen der mehrseitigen Auswahl. Die eine ist die mehrseitige Auswahl mit switch und die andere if mit else if.
Die erste Form mit switch ist sehr speziell, da die Bedingungsabfrage darauf beruht, dass sich die einzelnen Fälle nur in einem Punkt unterscheiden und sich alle auf einen Ausdruck beziehen der bei allen Fällen gleich ist. Das hört sich kompliziert an, ist jedoch einfach, wenn man sieht was gemeint ist. Beispiel Noten in der Schule: Bei allen ist der Ausdruck Note gleich, die einzelnen Fälle unterscheiden sich einzig und allein darin, dass der Wert der Note (1,2,3,4,5,6) unterschiedlich ist.

Mehrseitige Auswahl mit switch ... case

Im Gegensatz zu Select...Case in Visual Basic sind bei switch...case nur abzählbare Werte als Selektor (Fall) erlaubt, d.h. keine Datentypen wie double und float. Allgemein ist eine mehrseitige Auswahl mit switch...case so aufgebaut:

```
switch(Ausdruck)
{
 case wert1: Anweisung1;
 break;
 case wert2: Anweisung2;
```

break;
case wert3: Anweisung3;
break;
default: AnweisungN;

Wenn man sich die Quellcodezeilen also einzeln ansieht, dann lässt sich folgendes sagen: Man "nummeriert" die einzelnen Fälle nach folgendem Muster durch: case und dann der Wert z.B. Note 1, dann ist es wichtig, dass man den Doppelpunkt nicht vergisst. Dann kommt die eigentliche Anweisung, die mit einem Semikolon abgeschlossen wird. Wenn man Anweisungsblöcke mit mehreren Zeilen hat, dann sind diese in geschweifte Klammern zu stellen. Gerne vergessen wird dann das break-Kommando gefolgt von einem weiteren Semikolon. Lässt man dieses break-Kommando weg, dann wird der nächste Fall ebenfalls überprüft und möglicherweise ausgeführt (wenn man einen Zahlenraum angibt). Der default-Fall kann eigentlich weggelassen werden, er muss nur eintreten, wenn z.B. eine Meldung erscheinen soll wenn kein anderer Fall eingetreten ist. Hier ist kein break notwendig, da es immer der letzte Fall ist.
Wie schon bei if() muss darauf geachtet werden, dass hinter switch(Ausdruck) kein Semikolon zu stehen hat!

Mehrseitige Auswahl mit if()
Die Auswahlstruktur hatten wir schon einmal bei der ein- und zweiseitigen Auswahl. Mit dem Schlüsselwort else if kann man auch noch eine mehrseitige Auswahlstruktur daraus machen. Man unterscheidet einfach auch noch im if bzw. else-Fall. Also z.B.:
if(Bedingung) Anweisung;
else if (Bedingung2) Anweisung2;
else Anweisung3;

Wenn die erste Bedingung nicht erfüllt wird, dann schauen wir, ob die zweite Bedingung erfüllt wird, sonst führen wir die dritte Anweisung aus.
Bei dieser Möglichkeit besteht jedoch recht leicht die Gefahr, dass es sehr unübersichtlich mit den Fällen wird. Jedoch hat diese Methode den Vorteil, dass wir die freie Auswahl an Datentypen haben. Außerdem lässt sich so auf einfache Weise schnell mal eine if()-Auswahlstruktur erweitern, ohne dass man den Rest neu entwerfen müsste. Hierbei ist übersichtliche Gestaltung das A und O.

Wiederholungsstrukturen
Des öfteren ist es erforderlich, dass die gleichen Anweisung mehr als einmal ausgeführt werden müssen. Man könnte nun die Anweisungen manuell durch Mehrfacheingabe wiederholen. Das ist jedoch sehr umständlich und sehr oft nicht zweckmäßig. Was sollte man z.B. machen, wenn man solange etwas wiederholen möchte, bis der Benutzer eine bestimmte Taste drückt.
Für solche und andere Fälle gibt es Wiederholungsstrukturen. Diese wiederholen eingeschlossene Anweisungen abhängig von einer Bedingung (oder eben nicht).

Schleife mit Anfangsabfrage
Wie immer gibt es verschiedene Methoden Anweisungen zu wiederholen. Die erste ist die mit Anfangsabfrage, d.h. die Bedingung wird immer VOR der Ausführung der Wiederholung geprüft. Diese Schleifenart wird auch *kopfgesteuerte Schleife* genan-

nant, da die Anweisung im *Schleifenkopf* steht. Neben dem Schleifenkopf gibt es noch den Schleifenkörper, in dem die Anweisung(en) stehen.
Bei dieser Schleifenart kann es vorkommen, dass die Schleife nie ausgeführt wird, da die Schleifenbedingung von Anfang an nicht erfüllt wird. *Solange* die Schleifenbedingung aber von Anfang an *wahr* ist, solange wird die Schleife durchlaufen.
In der obigen Definition habe ich die Worte *solange* und *wahr* kursiv geschrieben. Diese Worte entsprechen nämlich ungefähr dem Java-Syntax für eine Schleife mit Anfangsabfrage. Die exakte Syntax sieht so aus:
while(Bedingung) Anweisung;

Zu deutsch also: Solange (Bedingung = wahr) Anweisung. Wie schon bei den vorhergehenden Kontrollstrukturen ist auch bei den Schleifen die mehrzeilige Anweisung in geschweifte Klammern zu schreiben.
Bei allen Schleifen besteht die Gefahr einer Endlosschleife. Bei der Endlosschleife ist die Bedingung falsch gewählt und zwar so, dass die Bedingung immer wahr ist und somit die Schleife unendlich oft wiederholt wird. Da der Benutzer keinen Einfluss mehr auf das Programm hat, dass immer weiter die gleichen Anweisungen ausführt, gilt das Programm als abgestürzt. Im schlimmsten Fall muss der Computer neugestartet werden. Aus diesem Grund ist es unbedingt erforderlich, dass die Schleifenbedingung genau überprüft wird, damit es nicht zu Endlosschleifen kommt.
Noch schlimmer ist nur noch, wenn man auch noch den Fehler begangen hat, ein Semikolon hinter while (Bedingung) zu setzen, wenn danach noch in geschweiften Klammern die Anweisungen stehen. Dann nämlich ist für den Benutzer nicht ersichtlich, was der Computer gerade macht. Da er immer unentwegt nichts macht, kommt es zu einem Hänger, der die gleichen Auswirkungen wie die Endlosschleife hat.

Schleife mit Endabfrage

Die zweite Möglichkeit der Wiederholung von Anweisungen ist die Schleife mit Endabfrage. Die Schleife wird auch fußgesteuerte Schleife genannt, da die Bedingung im Schleifenfuß steht. Die Schleife mit Endabfrage wird IMMER *mindestens einmal* durchlaufen. Die Schleife wird solange wiederholt, solange die Bedingung wahr ist. Somit ergibt sich folgende Java-Syntax:

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

Ansonsten gelten die gleichen Grundlagen wie bei der Schleife mit Anfangsabfrage.

Zählschleife

Die Zählschleife ist eine Schleife, bei der von Anfang an feststeht, wie viele Wiederholungen ausgeführt werden. Dabei wird die Anzahl der Wiederholungen von der Laufvariable (dem Zähler) gesteuert. Die Laufvariable erhöht sich bei jeder Ausführung um eine festgelegte Größe. Die Schleife wird solange wiederholt, bis der vorher festgelegte Endwert von der Laufvariable erreicht worden ist.
Die Syntax sieht etwas komplizierter aus, ist aber halb so schlimm:
for (Laufvariable=Anfangswert; Laufvariable<=Endwert; Laufvariable++) Anweisung;

Die Anweisung lässt sich in fünf Teile zerteilen. Der erste ist das Schlüsselwort for. In der Klammer stehen die nächsten drei jeweils getrennt durch ein Semikolon.
Erst kommt die Initialisierung der Laufvariable (Laufvariable=Anfangswert;), der Laufvariable wird der Anfangswert zugewiesen. Es folgt die Bedingung (Laufvariable<=Endwert;), bei der selbstverständlich die Laufvariable kleiner oder gleich dem Endwert sein muss. Ist das nicht mehr der Fall, wird die Schleife verlassen.
Die letzte in der Klammer stehende Anweisung (Laufvariable++) zeigt einen der Ursprünge von Java. Es handelt sich um einen sogennanten Postinkrementoperator, den der gebildete Programmierer aus C++ (gibt C++ seinen Namen) kennt. Dieser Postinkrementoperator tut nichts anderes, als die Laufvariable um eins zu erhöhen.
Nach der Klammer folgt dann selbstverständlich die Anweisung.

Arrays

Bis jetzt bestanden Variablen nur aus *einem* Wert, einer Zahl oder einem String. Arrays aber können eine *beliebige* Anzahl von Werten des *gleichen Typs* beinhalten. Bei der Erzeugung eines Arrays wird auch die Größe des Arrays festgelegt, d.h. wie viele Elemente gleichen Typs im Array gespeichert werden können. Das Array selbst besteht aus Feldern, diese sind mit dem sogenannten Index durchnummeriert und werden über den Index angesprochen. Der Index ist also die Adresse der Elemente im Array. Der Index beginnt mit null, und endet mit n-1. Dass heißt wenn man ein Array von der Größe sechs erzeugt, dann beginnt es mit null und endet mit fünf.
Das Array selbst wird, wie eine Variable auch, über seinen Namen angesprochen, da es aber viele Elemente (n + 1) des gleichen Typs enthält, wird zusätzlich noch der Index mit angegeben. Die Namensgebung des Arrays unterliegt den selben Beschränkungen wie eine Variable.
So jetzt wollen wir aber in die Java-Syntax einsteigen. So wird allgemein ein Array erzeugt:
Datentyp arrayName[];

Also z.B. *int noten[];*

So deklariert man zwar das Array, jedoch hat man es noch nicht dimensioniert. Das geht allgemein so:
arrayName = new Datentyp[Anzahl der Elemente];

Also z.B. *noten = new int[10];*

Wenn man es also ausführlich machen will sieht eine komplette Deklaration so aus:
int noten[];
noten = new int[10];

Man kann sich jedoch auch weniger Arbeit machen und das ganze aufeinmal machen:
int noten[] = new noten[10];

Mehrdimensionale Arrays

Es ist außerdem möglich, sogenannte mehrdimensionale Arrays mit Java zu erzeugen. Die Syntax ist der eines eindimensionalen Arrays sehr ähnlich. Die Syntax lautet:
Datentyp arrayName[] [] = new Datentyp [wert1] [wert2];

Also z.B. *int noten[] [] = new int [5] [7];*

Der Zugriff auf ein mehrdimensionales Array erfolgt genauso wie bei dem eindimensionalen Array mit dem Arraybezeichner und den Indices.

Exception Handling

In jedem noch so guten Programm kann auch mal ein Fehler auftreten (z.B. von außen: Papier von Drucker leer). Der Fehler ist nur noch halb so schlimm wenn man dafür eine gute und wirksame Fehlerbehandlung (Exception Handling) vorsieht.
In Java ist die Fehlerbehandlung nicht besonders schwierig. Fehleranfällige Programmteile werden von den Schlüsselwörtern *try* und *catch* umgeben. Das sieht ungefähr so aus:

```
try
{
  Anweisungen
}
catch()
{
  Anweisungen bei
  Fehlern
}
```

Wie zu sehen ist steht im Block nach *try* der Programmcode der auf Fehler zu überwachen ist und hinter *catch* folgt der Programmteil, der dann ausgeführt wird, wenn der Fehler abgefangen wurde. Es sind auch mehrere *catch*-Blöcke möglich. Im *catch*-Block muss die Art der Exception angegeben werden. Möglich Exception-Arten sind: *catch(StringIndexOutOfBoundsException e)* für Fehler die beim Zugriff auf Strings entstehen; *catch(NumberFormatException e)* für Fehler die beim Umwandeln in einen anderen numerischen Datentyp entstehen oder *catch(Exception e)* für alle anderen Fehler. Die mögliche Art der Exception lässt sich herausfinden, indem man nach den eingesetzten Anweisungen (z.B. *Integer.parseInt*) in der JDK-Dokumentation sucht. In der Beschreibung befindet sich auch die mögliche Exception Art. Nach der Exception-Art muss noch eine Variable stehen über die dann Informationen über den Fehler ausgegeben werden können (z.B. *e.toString()*).
Werden mehrere *catch*-Blöcke eingesetzt, so sollte immer der allgemeine catch-Fall (*catch(Exception e)*) zu letzt stehen, da alle *catch*-Fälle der Reihe nach aufgerufen werden bis der erste auf den Fehler zutrifft. Steht der allgemeine nicht am Ende, so kann man keine spezifische Fehlerbehandlung vornehmen.
Sollen Anweisungen im Fehlerfall nochmals wiederholt werden (z.B. Eingabe von Zahlen), so muss man diese in eine Endlosschleife packen, die könnte so aussehen:

```
while(true)
{
 System.out("Ganze Zahl: ");
 try
 {
    i = Integer.parseInt(in.readLine());
    break;
 }
```

```
catch (IOException e)
{
   System.out.println("IOException\n"+e);
   System.exit(0);
}
catch (NumberFormatException e)
{
   System.out.println("NumberFormatException\n"+e);
}
```

Mit *System.exit(0)* wird das Programm abgebrochen.
Das von dem ImputStreamReader bekannte throws IOException nach der main-Deklaration bewirkt, dass Fehler an die nächsthöhere Ebene weitergegeben werden. So braucht man pro Klasse nur ein Exception Handling zu schreiben. Wird in der nächst höheren Ebene ebenfalls kein Exception Handling durchgeführt, dann kommt es zu einer Laufzeitfehlermeldung durch den Interpreter.

Auswerfen von Exceptions

Bis jetzt sind Fehler immer nur vom Laufzeitsystem "geworfen" worden, man kann jedoch selbst Exceptions erzeugen! Wenn man z.B. nur Zahlen kleiner Zehn haben möchte kann man eine Exception erzeugen, wenn die Zahl größer als zehn ist. Dies geht mit dem Schlüsselwort *throw*. Eine Anweisung mit *throw* könnte so aussehen:
throw new SecurityException(zahl +" ist zu groß!");

Die Aufrufe müssen dann natürlich im *try*-Block stehen!

Eigene Exception-Klassen entwerfen

Man kann auch eigene Exception-Klassen definieren. Diese erben dann alle von der Klasse Exception. Eine eigene Exception-Klasse könnte Exceptions auslösen wenn eine negative Zahl eigegeben wird. Eine solche Exception könnte so aussehen:

```
public class NegativeIntegers extends Exception
{
   public NegativeIntegers()
   {
     super();
   }
   public NegativeIntegers(int i)
   {
     super ("Die Zahl " + i + " ist negativ! Das ist nicht erlaubt!");
   }
}
```

Die selbstentworfene Exception wird nun ganz normal in der Hauptklasse verwendet. Es wird dabei mit throws in der Methoden-Definition gearbeitet. Und eine Exception nach einer Überprüfung geworfen.

Lesen und schreiben von externen Dateien

Bis jetzt bestanden unsere Programme immer aus einer Eingabe von Hand oder war

bereits im Quelltext gemacht worden, das Ganze wurde dann auf dem Bildschirm ausgegeben. Es ist jedoch mit Java auch möglich externe Dateien einzulesen, bzw. in diese Ausgaben zu machen.
In der Eingabe von Hand ist schon ein Schlüsselwort eingebaut, das bei der Ein- und Ausgabe in Dateien mit Java eine große Rolle spielt den '*Stream*' (z.B. in In-put*Stream*Reader). Der Stream ist ein Datenfluss von der Eingabe hin zur Ausgabe, spielt also bei beidem eine Rolle.
In Java gibt es im Paket java.io.* eine Vielzahl von Streams (über 30); ich möchte vorerst nur den FileInputStream, den FileOutputStream, den FileWriter und den Buf-feredReader behandeln.

FileInputStream
Für einfache Dateieingaben benutzt man die Klasse *FileInputStream*. Man bindet mit dieser Klasse eine Datei an einen Datenstrom.
Allgemein sieht die FileInputStream-Anweisung so aus: ***FileInputStream(String)***
Damit wird ein FileInputStream mit einen gegebenen Dateinamen erstellt. Im Quell-text könnte das dann so eingesetzt werden:

```
[...]
FileInputStream in = new FileInputStream("Readme.txt");
int zeichen =0;
while ((zeichen=in.read())!= -1)
System.out.print((char)zeichen);
in.close();
[...]
```

Hier sieht man warum der FileInputStream nur für einfache Dateieingaben verwendet wird. Er liest Byte für Byte aus der Datei aus und liefert int-Werte (Die dem ASCII-Key entsprechen) zurück. Da alles Byte für Byte geschieht, muss man eine Schleife erstellen, die den Vorgang solange wiederholt bis die Datei zu Ende ist (ASCII-Key -1 bedeutet Dateiende). Für die Ausgabe müssen die int-Werte mittels Typecast ((char)zeichen) in Buchstaben umgewandelt werden.
Als Exceptions können FileNotFoundException und *IOException* auftreten. Ersteres ist als Datei nicht vorhanden und letzteres als Lesefehler zu interpretieren.

FileOutputStream
Der *FileOutputStream* ist das Gegenstück zum *FileInputStream*. Auch er ist nur für einfache Dateiausgaben geeignet.
Allgemein wird so ein *FileOutputStream* aus einem gegebenen Dateinamen erzeugt: ***FileOutputSteam(String);***
Im Quelltext könnte man ihn z.B. so einsetzen:

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
FileOutputStream fileOut = new FileOutputStream("line.txt");
System.out.println("Text eingeben:");
String text = in.readLine();
fileOut.write(text.getBytes(), 0, text length());
fileOut.close();
```

Hierfür musste ich etwas ausholen, zuerst muss natürlich etwas eingegeben werden, dies geschieht über den *BufferedReader*. Dieser Text wird in einer String-Variable gespeichert und mit der *write*-Methode des FileOutputStream-Objektes in eine Datei

geschrieben.
Wie im Quelltextschnipsel zu sehen ist, sind Parameter zwingend erforderlich! Die *write*-Methode ist also folgendermaßen aufgebaut: *write(Daten, Anfang, Länge)*
Daten sind ein Array, jedoch kein String-Array, sondern ein Byte-Array, aus diesem Grund werden die Daten mittels *.getBytes()* ins Byte-Format konvertiert; der *Anfang* ist bei 0 wenn die Datei noch leer ist und die *Länge* wird mittels *.length()* bestimmt.

FileWriter
Wesentlich einfachergeht das Schreiben in Dateien aber mit dem FileWriter. Er ermöglicht das sofortige Schreiben in Dateien. So sieht die Anweisung in Aktion aus:
```
FileWriter fw = new FileWriter("fileWriter.txt");
fw.write("Testzeile in Textdatei");
fw.close;
```

Mit den hier gezeigten Einstellungen wird eine Datei erzeugt, besteht diese schon, so wird sie gelöscht. Möchte man, dass die Textzeilen hinten angehängt werden, dann muss man einen zweiten Parameter setzen, das sieht dann so aus:
```
FileWriter fw = new FileWriter("fileWriter.txt", true)
```

Es reicht also schon ein *true* anzuhängen und schon ist auch dieses Problem gelöst. Es müssen nur IOExceptions abgefangen werden.

BufferedReader / FileReader
Der *BufferedReader* ist ein alter Bekannter aus der Eingabe über Tastatur. Er ist jedoch so universal einsetzbar, dass er auch aus Dateien lesen kann. Wenn wir uns nochmals an die *BufferedReader*-Deklaration erinnern wollen:
```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

So nun vergleich wir das mit dieser Funktion die für das Einlesen aus Dateien zuständig ist:
```
BufferedReader in = new BufferedReader(new FileReader("Textdatei.txt"));
```

Wir erkennen, dass sich die zwei Anweisungen nur durch den Parameter unterscheiden. Beim Einlesen von Textdateien ist dieser der *FileReader*.
Auch der Einsatz des *BufferedReaders* hat Vorteile gegenüber dem Einsatz des FileInputStreams. Er kann Zeilenweise einlesen! Das macht ihn natürlich sehr viel schneller.
So könnte man ihn im Quelltext verwenden:
```
[...]
BufferedReader in = new BufferedReader(new FileReader("Textdatei.txt"));
String str;
while((str=in.readLine())!=null)
System.out.println(str);
in.close;
[...]
```

Als Exceptions können *FileNotFoundException* und *IOException* auftreten.

Abschluss
Damit sind die Grundlagen der Programmierung mit Java abgeschlossen, ich empfeh-

le jedoch für anspruchsvolleres Programmieren auch die Workshops zu Klassen und der Grafikprogrammierung mit dem AWT.