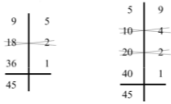
1 Ein Algorithmus und seine Implementierung in Java

Beispiel-Algorithmus: altägyptische Multiplikationsmethode:
Effizient für Computer im Dualsystem:

Verdoppeln → „left shift"
00101
01010

Halbieren → „right shift" (abgerundet)
00101
00010



Problemreduktion: Das Problem muss auf sinnvolle Weise vereinfacht, reduziert werden. Diese Reduktion hat irgendwann ein Ende, das ebenfalls festgelegt werden muss (in diesem Fall $b = 1$).

$$f(a,b) = \begin{cases} a & falls\ b = 1 \\ f\left(2a, \frac{b}{2}\right) & falls\ b\ gerade \\ a + f\left(2a, \frac{b-1}{2}\right) & sonst \end{cases}$$

Rekursive Funktion implementiert:

```
static int f(int a, int b){
  system.out.println(a + "   " + b);
  if (b == 1)    return a;
  if (b%2 == 0) return f(a+a, b/2);
     else return a + f(a+a, (b-1)/2);
}
```

Das interessiert uns jetzt nicht
Die Methode heisst „f"
Das Ergebnis und die beiden Parameter sind ganze Zahlen
Zugehörige schliessende Klammer „}" am Ende
Liefert der Rest bei der ganzzahligen Division
Jedes Mal, wenn die Methode f „aufgerufen" wird, schreibt sie die Parameter auf das Display

Zum Testen und Finden von Fehlern sollten immer Spezialfälle wie $f(0,1), f(1,0)$, usw. getestet werden. Bei $f(1,0)$ gibt es beispielsweise eine endlose Rekursion, da die Abbruchbedingung nie eintritt.

Arithmetischer Überlauf: Sind die Zahlen, die multipliziert werden sollen, zu hoch, kommt es zu einem Überlauf. Dies passiert, weil die Zahl grösser wird, als im „darstellbaren Bereich" möglich ist (Für `int` -2147483648 bis 2147483647).

Prüfung der Korrektheit (induktiv):
Behauptung: ∀ a, b ∈ ℕ⁺: f(a,b) = a × b.

Beweis induktiv über b:
- b = 1:
  ∀ a ∈ ℕ⁺: f(a,1) = a = a × 1 (gilt offensichtlich nach der Def. von f, *Fall 1*)
- b = n+1, mit der Induktionsannahme ∀ a ∈ ℕ⁺, ∀ b ∈ {1,...,n}: f(a,b) = a×b:

a) Sei b gerade: Es gilt
f(a,b) = f(2a,b/2) [nach Definition, *Fall 2*]
= 2a × b/2 [wg. Induktionsannahme, da b/2 ∈ {1,...,n}]
= a × b.

b) Sei b ungerade (und ≠1): Es gilt
f(a,b) = a + f(2a, (b-1)/2) [nach Definition, *Fall 3*]
= a + 2a × (b-1)/2 [wg. Induktionsannahme da (b-1)/2 ∈ {1,...,n}]
= a + a × (b-1)
= a × b.

Aber wieso eigentlich?

Exceptions: Ausnahmen sind Fehler, die oft vom System ausgelöst werden. Sie können abgefangen / behandelt werden.

```
static int f(int a, int b) {
  if (b == 1) return a;
  try
  {
    if (b%2 == 0) return f(2*a, b/2);
       else return a + f(2*a, b/2);
  }
  catch (StackOverflowError e)
  {
    ...
    System.out.println("..."...a...b...);
    System.exit(1);
  }
  …
}
```

Hier erwarten wir keine Fehler, daher ausserhalb des try-Blocks
e könnte man benutzen, um mehr über den Fehler zu erfahren
Notausstieg aus dem Programm – könnten wir statt dessen etwas sinnvolleres tun?

Es gibt auch Fehler die nicht abgefangen werden können, wie z.B. einen arithmetischen Überlauf.

Invariante als Beweistechnik: Das Programm kann auch anders realisiert werden:

```
static int f(int i, int j)
  int a = i;
  int b = j;
  int z = 0;
  while (b > 0) {
    if ungerade(b) {
      z = z+a;
      b = b-1;
    }
    b = b/2;
    a = 2*a;
  }
  return z;
}
```

i und j werden in der Methode nicht verändert
Hier gilt offenbar a x b + z = i x j
Die Methode `ungerade(b)` kann man als einfache Übung implementieren
Falls hier a x b + z = i x j
dann auch hier
Falls hier a x b + z = i x j
dann auch hier
Hier gilt offenbar b=0 und a x b + z = i x j
Also gilt hier z = i x j
Also wird **i x j** zurückgeliefert!
z akkumuliert die Werte der linken Spalte nicht gestrichener Zeilen
Hier ist b gerade
Zur Übung: Geht b/2 bzw. 2*a auch mit den Shift-Operatoren `b >>= 1` bzw. `a <<= 1`?
a x b + z ist eine **Invariante**

Der Ausdruck $a \cdot b + z$ ist eine *Invariante*. Er bleibt stets unverändert, auch wenn die einzelnen Werte sich ändern. Damit ist klar: $a \cdot b + z = i \cdot j$.

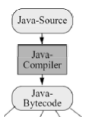
Effizienz des Algorithmus: Kann an der Gesamtzahl der elementaren Operationen gezählt werden, die pro Aufruf auftreten können. Wesentlich ist weiter noch die Anzahl rekursiver Aufrufe. Man könnte die Effizienz noch genauer untersuchen, indem man verschiedenen Operationen als „teurer" oder „billiger" definiert. $\frac{b}{2^x} \leq 1 \Rightarrow x \geq \log(b)$. Es werden also nicht mehr als $5 \cdot \log_2(b)$ Operationen benötigt (Logarithmus immer Basis 2).

```
static int f(int a, int b){
  if (b == 1)  return a;
  if (b%2 == 0) return f(a+a, b/2);
     else return a + f(a+a, b/2);
}
```
1 2 3 4 5

Wichtig ist hierbei auch der Vergleich mit ähnlichen Algorithmen, die zum gleichen Ziel führen können.

2 Java

Allgemeines: Java und C++ bilden eine gemeinsame Sprachfamilie, d.h. die Syntax ist einheitlich und die Semantik analog. Java ist generell moderner und konsequenter. Es werden ausserdem viele vorgefertigte Pakete zur Verfügung gestellt, was das Programmieren sehr vereinfachen kann.



Der Java-Bytecode, das vom Java-Compiler erstellt wird kann von einer virtuellen Maschine gelesen werden. Diese ist ein programmierter Simulator eines abstrakten Prozesses. Es kommt durch diese Interpretation zu einem Effizienzverlust.

Programmstruktur:
Mit `import` werden vorhandene *Pakete* von Klassen verfügbar gemacht. Der *Klassenkörper* enthält Instanzen- und Klassenvariablen, Konstanten und klassenbezogene `static`-Methoden. *Methoden* stellen Funktionen bzw. Prozeduren dar. *Konstruktoren* sind spezielle Methoden, die eine Klasse erzeugen. Methoden haben Namen und bestehen aus Parametern, lokalen Variablen und Anweisungen. Bei eigenständigen Programmen muss es eine `main`-Methode geben. Jede Klasse kann eine solche Methode enthalten, das Programm wird bei Aufruf der Klasse hier gestartet. Variablen im Klassenkörper sind global sichtbar innerhalb der Klasse.

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

Einfache Datentypen:

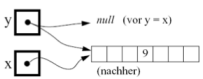
Name	Bytes	Wertebereich
bool	1	false, true
char	2	65536 versch. Werte
int	4	-2147483648...2147483647
byte	1	$-2^7 \dots 2^7 - 1$
short	2	$-2^{15} \dots 2^{15} - 1$
long	8	$-2^{63} \dots 2^{63} - 1$
float	4	3,4E-38...3,4E38
double	8	1,7E-308...1.7E308

Konventionen für Namensdeklarationen:
Variablen und Methoden:
`int i; public ausgabe (...)`
Klassennamen:
`class Person {...}`
Konstanten:
`MAX_SIZE`

Arrays: Mathematisch betrachtet endliche Folgen:

```
int [] x; // array of int
x = new int[ 7]; // Grösse 7 (Indexbereich 0..6)
for (int i=0; i < x.length; i++) x[ i] =17;
int [] x = new int[ 7]; // so ginge es auch
int [] y;
y = x;      // y zeigt auf das gleiche Objekt
y[ 3] = 9;  // x[ 3] ist daher jetzt auch 9
```

Grösse, d.h. Anzahl der Elemente



Arrays sind Referenzen auf (Speicher)-Objekte: Vorsicht bzgl. der Kopiersemantik („Aliaseffekt") und beim Vergleich zweier Array-Variablen!

```
float [][] matrix = new float[ 4][ 4];
matrix[ 0][ 3] = 2.71;
```

Da Arrays mit `new` dynamisch erzeugt werden, kann die Grösse eines Arrays zur Laufzeit bestimmt werden. Auch mehrdimensionale Arrays sind möglich.

Typkonversion: Java ist streng typisiert. Dies kann jedoch durchbrochen werden. Man benutzt eine *explizite Typenumwandlung*.

```
int myInt;
double myFloat = -3.14159;
myInt = (int)myFloat;
```

Hüllenklassen: Einfache Datentypen sind keine Objekte, für diese gibt es Hüllenklassen. Objekte davon können überall verwendet werden, wo eine Objektreferenz verlangt wird. Sie bieten einige sehr nützliche Methoden und Attribute.

```
int x = 5;  // normaler "int"
Integer iob = x;  // Instanz der Klasse "Integer"
if (iob == 5) then ...  // sind typkompatibel
```

Ein- und Ausgabe: `System.in` stellt den Standard-Eingabestrom dar.

```
int count = 0;
while (System.in.read() != -1) count++;
System.out.println("Eingabe hat " +
                   count + "Zeichen.");
```

`System` ist eine Klasse mit Schnittstellenmethoden. `read` liest ein einzelnes Zeichen und liefert `-1` bei Dateiende. `System.out` ist der Standard-Ausgabestrom. `print` gibt das übergebene Argument aus, `println` erzeugt danach noch einen Zeilenumbruch. Alle einfachen Datentypen können hier ausgegeben werden.

Beispiel: Einlesen von Zahlen:

In diesem Paket stehen die Ein-Ausgabe-Methoden

Die auftretbaren Exceptions müssen nach `throws` am Anfang einer Methode genannt werden

```
import java.io.*
class X {
  public static void main(String args[])
  throws java.io.IOException {
    int i=0; String zeile;
    DataInputStream ein = new DataInputStream(System.in);
    while(true) {
      zeile = ein.readLine();
      i = i + Integer.parseInt(zeile);
      System.out.println(i);
    }
  ...
  }
  ...
}
```

Der Eingabestrom muss beim Aufruf des Konstruktors angegeben werden

Die Klasse DataInputStream enthält die Methode readLine, welche alle Zeichen bis Zeilenende liest und daraus einen string konstruiert

„parseInt" ist eine Methode der Klasse „Integer", die einen string in einen int-Wert konvertiert (analog kann man auch Gleitpunktzahlen einlesen)

Man könnte hier auch `ein.readLine()` für `zeile` substituieren

Beachte: Die Methode readLine kann eine IOException auslösen!

Strings: Zeichenketten werden von der Standardklasse `String` realisiert, sie sind keine Arrays of `char`.

```
String msg = "Die"; // String-Objekt wird
int i = 7;          // automatisch erzeugt

//msg = new String("Die"); // So ginge es auch
msg = msg + " " + i;   // Konkatenation
msg = msg + " Zwerge";

System.out.println(msg); // Die 7 Zwerge
System.out.println(msg.length()); // 12

String b = msg;
msg = null;
System.out.println(b); // Die 7 Zwerge
```

Strings können auf verschiedene Weisen verglichen werden: Der Vergleich mit `==` stellt einen Referenzvergleich dar. Ein Wertevergleich wird mit `s1.equals(s2)` erreicht. Mit `s1.compareTo(s2)` prüft man die lexikographische Anordnung (Länge). Es gibt eine Vielzahl weiterer Methoden.

3 Klassen und Referenzen

Erstellen einer Klasse anhand eines Beispiels:

`private`-Attribute sind ausserhalb der Klasse nicht sichtbar. Eine Methode mit dem gleichen Namen wie die Klasse heisst *Konstruktor*. Beim Instanziieren eines Objekts einer Klasse wird dieser als erstes aufgerufen. Der erste stellt einer default-Konstruktor dar, da er keine Argumente erhält. Welcher Konstruktor genommen wird, richtet sich nach der Signatur beim `new`–Aufruf. Da man auf die `private`–Attribute nicht zugreifen kann, werden „Accessor-" oder „Mutator"-Methoden benutzt, wie `Drucken` oder `Setzen`.

```
class Datum {
  private int Tag, Monat, Jahr;
  public Datum() {
    System.out.println("Datum mit Wert
                        0.0.0 gegründet");
  }
  public Datum(int T, int M, int J) {
    Tag = T; Monat = M; Jahr = J;
  }
  public void Drucken() {
    System.out.println(Tag + "." + Monat
                       + "." + Jahr);
  }
  public void Setzen(int T, int M, int J) {
    Tag = T; Monat = M; Jahr = J;
  }
}
```

Man hätte hier auch präziser sein können: this.Tag...; this.Monat etc.

Verwendung:
Ostermontag ist eine *Referenz*, die auf Datum-Objekte *zeigen* kann. Man kann die Klasse beliebig erweitern: Hier gibt es einen Unterschied zu C++: Operatoren wie ‚<' können hier *nicht überladen* werden!

```
class Beispiel {
  public static void main (String args[]) {
    Datum Ostermontag = new Datum();
    /* ==> Datum mit Wert 0.0.0 gegründet */
    Ostermontag.Drucken();
    Ostermontag.Setzen(09,04,2012);
    Ostermontag.Drucken();
    ...
  }
}
```

Hier wird der erste Konstruktor aufgerufen

Liefert 0.0.0

Liefert 9.4.2012

Für Gleichheitsvergleiche gilt dasselbe wie für `String`-Objekte: ein ‚==' vergleicht die Referenzen, nicht die Werte.

```
class Datum {
  ...
  public boolean frueher_als(Datum d) {
    return Jahr < d.Jahr ||
           Jahr == d.Jahr && Monat < d.Monat ||
           Jahr == d.Jahr && Monat == d.Monat
                          && Tag < d.Tag;
  }
```

true oder *false*

d ist ein formaler Parameter vom Typ „Datum"

Man kann auf d.Jahr, d.Monat, d.Tag zugreifen, obwohl diese Attribute als „private" deklariert sind, da es sich um die *gleiche Klasse* (aber eine andere *Instanz*) handelt

`this` ist ein Schlüsselwort, mit dem stets eine Referenz auf das eigene, aktuelle Objekt zurückgeliefert wird.

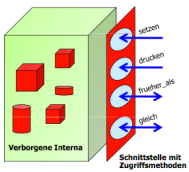
```
class Datum...
  boolean frueher_als...
  boolean gleich(Datum d) {
    return !frueher_als(d) && !d.frueher_als(this);
  }
...
```

Objekt d aufgerufen, und zwar mit „einem selbst" als Parameter!

„gleich" liesse sich

```
class Datum {
  private int Tag, Monat, Jahr;
  static String Monatsname(Datum d)
    if (d.Monat == 1) return "Januar";
    if ...                   "Februar";
      ...
    }
  ...
}
```

Klassen- und Instanzenmethoden: Klassenmethoden bekommen das Attribut `static` vorangestellt. Sie werden nicht für spezifische Objekte aufgerufen, sondern für die Klasse als Ganzes. Sie lösen somit allgemeine Aufgaben für alle Klassen. Klassenmethoden können nicht auf spezifische Instanzenvariablen zugreifen und auch nicht auf die `this`-Referenz. Sie können nur auf `static`-Variablen und andere Klassenmethoden zugreifen. Der Zugriff auf Klassenmethoden erfolgt durch die Angabe des Klassennamens:
`System.out.print(Datum.Monatsname(x));`
Klassenbezogene Variablen (`static`) werden von allen Instanzen geteilt. Eine Änderung wirkt für alle.



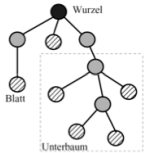
Information Hiding: Dient der *Kapselung* aller relevanten Daten und Methoden. Zugriff auf private Daten erfolgt nur über Zugriffsmethoden. Die interne Repräsentation ist nach aussen unsichtbar, was die Integrität garantiert. Dies ermöglicht eine konsequente Realisierung abstrakter Datentypen und bietet eine änderungsfreundliche Software.

Dynamische Klassen und Referenzen: Das Erzeugen von dynamischen Klassen geschieht mit `p1=new x();`, wobei `x()` den Konstruktor darstellt. Es wird eine Instanz der Klasse `x` erzeugt und eine *Referenz (Adressen)* zurückgeliefert. Zeigt eine Referenz nirgendwo hin, zeigt sie auf `null`. Sie können einander zugewiesen werden, wobei jeweils neue Zeiger entstehen:
`Person p, q; p = new Person(); q = p;`

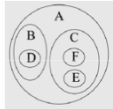
4 Syntaxanalyse und Compiler

Baum, Wurzelbaum, Binärbaum: Bäume bestehen aus *Knoten* und *Kanten* und haben folgende Eigenschaften:

(i) Jede Kante verbindet 2 Knoten.
(ii) Zwischen 2 Knoten gibt es höchstens eine Kante.
(iii) #Knoten = #Kanten + 1.
(iv) Bäume sind zusammenhängend und azyklisch.
(v) Zwischen 2 Knoten gibt es genau 1 Weg.



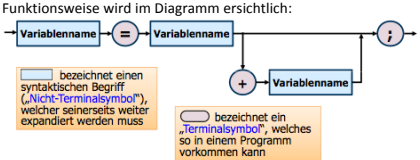
Bei *Wurzelbäumen* wird ein bestimmter Knoten als *Wurzel* betrachtet, von dem aus der Baum „wächst". Die „letzten" Knoten eines solchen Baums werden *Blätter* genannt. Man kann die Knoten in *Niveau, Ebenen* aufteilen (Entfernung zur Wurzel). Jeder Knoten kann als Wurzel eines Unterbaums aufgefasst werden. Verschiedenste Darstellungen eines Baums sind möglich.



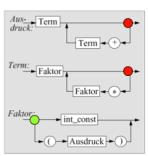
Binärbäume sind Bäume, bei denen jeder Knoten höchstens 2 Nachfolger hat, wobei man noch vom linken und rechten unterscheiden kann. Diese können gut in Arrays dargestellt werden (sehr einfach), wobei Lücken ordentlich kodiert werden müssen. Die Idee besteht darin, die Wurzel an die Stelle 1 (nicht 0) zu speichern und die beiden Nachfolger der Stelle i an die Stellen $2i$ und $2i+1$.

	4	6	9	12	13	11	17	18	16	19	22	32
0	1	2	3	4	5	6	7	8	9	10	11	12

Syntaxdiagramme: Zur Syntaxanalyse werden sogenannte *Syntaxbäume* verwendet. Sie spiegeln die Programmstruktur wieder. Dies dient dem Zweck syntaktische Fehler zu finden. Die Funktionsweise wird im Diagramm ersichtlich:



Syntaxanalyse durch rekursiven Abstieg: Jedes Teildiagramm eines Syntaxdiagramms durch eine Methode realisieren, welche das entsprechende Konstrukt versteht. Ein *Parser* (Zerteiler) ist ein Syntaxanalyseprogramm:




```
//Anfangsteil kommt später
  ...
void int_const(){
  c = KbdInput.getc();
}

void Ausdruck(){
  Term();
  while (c == '+') {
    c = KbdInput.getc();
    Term();
  }
...
```

Integer-Konstanten bestehen bei uns nur aus einem einzigen Zeichen

Verarbeitetes Zeichen „weglesen"; nachfolgendes Zeichen einlesen

Bemerkung: KbdInput ist eine Klasse mit Methoden (wie z.B. „getc"), die nicht zum Standard-Java gehört. Die Funktionalität (Zeichen vom Keyboard lesen) davon lässt sich aber leicht mit Methoden aus dem Paket java.io.* nachbilden (Übung!)

```
//Anfangsteil kommt später
  ...
void int_const(){...}
void Ausdruck(){...}
void Term() {
  Faktor();
  while (c == '*') {
    c = KbdInput.getc();
    Faktor();
  }
}
void Faktor() {
  if ((c >='0') && (c <='9'))
    int_const();
  else
    if (c == '(') {
      c = KbdInput.getc();
      Ausdruck();
      if (c == ')')
        c = KbdInput.getc();
      else Fehler(1);
    }
    else Fehler(2);
}
```

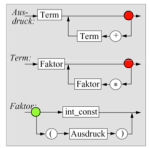
"Term an Faktor: 'Lies einen Faktor!'"

"Faktor an int_const: 'Lies eine int_const!'"

Rekursion!

```
public static void main(String args[])
{
  Parser p = new Parser();
  p.c = KbdInput.getc();
  p.Ausdruck;
}
```

"Hauptprogramm an Parser p: 'Lies einen Ausdruck!'"




```
import packagexyz.KbdInput;

class Parser() {
  char c;

  void Fehler(int f) {
    switch f {
      case 1:
        System.out.println("Fehlende Klammer ')'");
        break;
      case 2:
        System.out.println("Falsches Zeichen");
        break;
      default:
        System.out.println("Interner Fehler");
        break;
  }
  //...
  //Die anderen Methoden
}
```

Dort sei die Methode getc definiert, die ein Zeichen als 'char' einliest

global für alle Methode

Kapselung aller natürlichsprachlichen Meldungen in einer Methode → Anpassung an eine andere Sprache etc.

Defensives Programmieren: mit falschem Parameter rechnen!

Wenn nach Abarbeitung der gesamten Eingabe kein Fehler gemeldet wurde, dann lag ein (gemäss den Syntaxdiagrammen) korrekter Ausdruck vor ⇒ Programm ist ein „Syntaxchecker"

Erzeugung eines Syntaxbaums: Bei korrekter Syntax soll ein Syntaxbaum generiert werden, in welchem die Ausdrücke stehen (eingerückte Form):

```
void out(int t, String x) {
  System.out.println();
  for(int i=1;i<=t;i++)
    System.out.print(" ");
  System.out.print(x);
}
```

```
void Ausdruck(int t) {
  out(t,"Ausdruck");
  Term(t+1);
  while (c == '+') {
    out(t+1,"+");
    c = KbdInput.getc();
    Term(t+1);
  }
}

void int_const(int t) {
  out(t,"int_const: " + c);
  c = KbdInput.getc();
}
```

"out" gibt den Namen entsprechend eingerückt aus

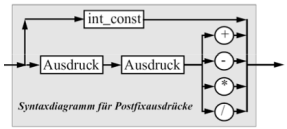
"Term" und "+" haben als Nachfolger von Ausdruck eine um 1 grössere Tiefe

Damit wir den Wert der Konstanten erfahren

```
(1+2)*3
Ausdruck
  Term
    Faktor
      (
      Ausdruck
        Term
          Faktor
            int_const: 1
        +
        Term
          Faktor
            int_const: 2
      )
  *
  Faktor
    int_const: 3
```

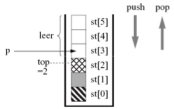
Binäre Operatorbäume: Entfernt man unnötiges, lässt sich der Syntaxbaum stark reduzieren. Man kann die Operatoren als Wurzel eines Unterbaums sehen. Die Blätter stellen Zahlen dar. Prinzip: Zuerst den linken Unterbaum traversieren, dann den Wert der Wurzel ausgeben, anschliessend den rechten Unterbaum traversieren. Dies generiert den vollständig geklammerten Ausdruck (infix).

Postfix-Ausdrücke: Bei Postfix-Ausdrücken kommt der Operator nach den zugehörigen Operanden, nicht dazwischen (infix).



Syntaxdiagramm für Postfixausdrücke

Für die Umwandlung von der infixen zur postfixen Darstellung nutzt man einen *Stack*:



- Für einzelne Zeichen („character-Stack")
- Realisiert als Klasse, die ein Array enthält
- Array-Grenzen sind durch 0 und length-1 abgesteckt
- p „zeigt" immer schon auf das nächste freie Element

```
class Stack {
  int p;                          Stackpointer
  char [] st;
  Stack(int size) {               Konstruktor
    p = 0;
    st = new char[ size] ;
  }
  void push(char c) {             Sonst Fehler bei Zugriff auf st[st.length]
    if (p >= st.length)
      System.out.println
       ("Stack Overflow");
    else
      st[ p++] = c;
  }
  char pop() {
    return st[ --p] ;             Ein "Stack Underflow" sollte eigentlich auch überprüft werden!
  }
}
```

```
class Stack //Wie gehabt...
class InfToPost {
  Stack stk = new Stack(1000);
  char c;

  boolean eof(char c) {
    return (c == (char) -1);
  }

  void convert() {
    while (!eof(c = KbdInput.getc())) {
      if ((c=='+') || (c=='*'))
        stk.push(c);
      if ((c >= '0') && (c <= '9'))
        System.out.print(" " + c);
      if (c == ')')
        System.out.print(" " + stk.pop());
    } //end while
  System.out.println();
  }
} //end class InfToPost
```

Unverständlich geklammerte Ausdrücke um?

Bei Dateiende ("end of file") wird ein Spezialzeichen geliefert, das dem Wert (char)-1 entspricht

Als Seiteneffekt bei der Zuweisung wird der Wert an eof weitergegeben

Ziffern direkt ausgeben

Ein Testbeispiel:
(5*((1+2)*(3*4))+6))
wird umgewandelt in
5 1 2 + 3 4 * * 6 + *

Postfix-Auswerter:

```
public static main void(String [] )
Stack stk = new Stack(1000); // hier: int-Stack (für Operanden)
  char c = ' '; int x;
  while (!eof(c)) {
    if (! (c == '+' || c == '*' || c >= '0' || c <= '9')) {
      c = KbdInput.getc();                  Alles Fremde einfach überlesen; z.B. auch Leerzeichen, newline, ...
      continue;
    }
    if (c == '+') {
      stk.push(stack.pop() + stack.pop());  Die Operation '+' oder '*' werden auf die beiden obersten Stackelemente angewendet, das Ergebnis davon gleich wieder auf den Stack gepusht
      c = KbdInput.getc();
      continue;
    }
    if (c == '*') {
      stk.push(stack.pop() * stack.pop());
      c = KbdInput.getc();
      continue;
    }
    x = 0;
    while (c >= '0' && c <= '9') {          Mehrziffrige Operanden zusammenbauen und den berechneten int-Wert auf den Stack legen
      x = 10 * x + (c - '0');
      c = KbdInput.getc();
    }
    stk.push(x);
  }
  System.out.println(stk.pop());            Ausgabe des Endergebnisses
}
```

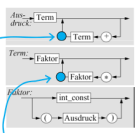
Rekursiver Klammerchecker:

```
class KlammerChecker {
  //c und zeile sind global für alle Methoden
  char c; int zeile = 1;                    Erzeugt gleiches Ergebnis wie vorheriges Programm
  static boolean eof(char c) { return c == (char) -1; }

  void block(int x) {                       Implizite Nutzung des Laufzeitstacks (statt explizitem Stack)
    while (!eof(c = KbdInput.getc())) {
      switch(c) {
        case '\n':
            zeile++;                        Ginge auch continue?
            break;                          Die Zeilennummer wird dabei auf dem jeweiligen Exemplar der Variablen z der einzelnen Methoden-instanzen gespeichert
        case ')':
            System.out.println("Block von zeile " + x +
                               "endet in Zeile" + zeile);
            return;
        case '(':
            block(zeile);                   Rekursiver Aufruf!
      } // end switch
    } //end while
  }

  public static void main(String[] args) {  Nach Rückkehr aus einem rekursiven Methodenauf-ruf befindet man sich wie-der in einer Instanz mit dem „alten" Wert von z
    KlammerChecker kc = new KlammerChecker();
    kc.block(0);                            Übung: Wie könnte man Fehler der Klammerstruk-tur (zu viele bzw. zu wenige '(' oder ')') melden?
  }
}
```

Codegenerierung für Infix-Ausdrücke: Aufgabe des Compilers ist neben der Syntaxprüfung auch „Code" für die Zielsprache zu Erzeugen. Das Analyseprogramm kann hierfür an den richtigen Stellen mit Codeerzeugungshinweisen ausgestattet werden.

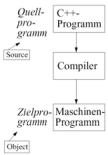
```
void int_const(){ //hier nur einziffrig
  System.out.println("push(" + c + ")");
  c = KbdInput.getc();
}

void Ausdruck(){
  Term();
  while (c == '+') {
    c = KbdInput.getc();
    Term();
    System.out.println("plus");
  }
}

void Term() {
  Faktor();
  while (c == '*') {
    c = KbdInput.getc();
    Faktor();
    System.out.println("mult");
  }
}
```

Code-Generierung für die Stackmaschine



Auch bei dieser Infix zu Postfix-Übersetzung wird (implizit) ein Stack benutzt: Der Laufzeitstack von Java, in dem die Rücksprung-adressen abgelegt sind!

Compiler: Übersetzt ein Programm einer Programmiersprache ganz in Maschinensprache. Ein Prozessor kann nur Maschinensprache ausführen. Die Transformation muss bedeutungsäquivalent sein.




```
for(int i=1;i<=n;i++)
  x = 2*y + x ;
```

```
     MVC 0,I
LOOP CMP N,I
     JEQ FIN
     ADC 1,I
     LDV X,(1)
     LDV Y,(2)
     SLL (2),1
     ADR (2),(1)
     STO (1),X
     JMP LOOP
FIN ...
```

Interpreter: Programm, welches ein Programm einer anderen Sprache Anweisung für Anweisung intern übersetzt und unmittelbar ausführt.

Java-Bytecode: Ist die Maschinensprache der Java-VM. Bytecode ist sehr kompakt: die meisten Instruktionen sind 1 Byte lang. Die Speicherplätze für Variablen werden vom Compiler durch-nummeriert (Adressen).

```
static int p1(int p) {
  int i;
  i = 5;
  int j = 7;
  int k = j+i+j+i+3;
  return k;
}
```

```
0 iconst_5
1 istore_1
2 bipush 7
4 istore_2
5 iload_2
6 iload_1
7 iadd
8 iload_2
```

5 Pakete in Java

Ein *Paket* bindet zusammengehörige Menge von Klassen, sowie Unterpakete und Interfaces. Sie sind hierarchisch aufgebaut: Paket „xyz" im Paket „java" → „java.xyz". Sie sind wichtig für die Strukturierung und Zugriffskontrolle, denn Klassen und Methoden sind default nur im eigenen Paket sichtbar. Klassen befinden sich immer in Paketen. Pakete werden immer am Anfang einer Quelldatei deklariert und werden stets mit Kleinbuchstaben benannt: `package abc;`
Attribute und Methoden von Klassen können *vollqualifiziert* benannt werden: `java.lang.String.substring`
Das *Importieren* von Paketen erfolgt folgendermassen:
`import java.util.Random / java.util.*`

Beispiel: `int`-Listen (linked List)

```
package listPack;

class ListElem {                    ListElem ist ausser-halb des Paketes nicht sichtbar
  int val;
  ListElem next;

  public ListElem(int i, ListElem e) {   Konstruktor
    val = i;
    next = e;                       Hierauf soll das neue Listenelement zeigen
  }
}

public class List {                 Nur eine einzige Klasse pro Datei darf public sein
  ...
}
```

```
public class List {
  private ListeElem first = null;
  private int size = 0;

  private boolean empty() {
    return (size == 0);
  }

  public int size() { return size;}

  public void add_head(int i) {
    first = new ListElem(i,first);
    size++;
  }

  public int remove_head() {
    int i = first.val;
    first = first.next;
    size--;
    return i;
  }

  // evtl. weitere Methoden
```

Kein Zugriff von aussen

Methode liefert den Wert der gleichnamigen Variablen

Neues Element vorne einketten

Grösse auf privater Variablen mithalten

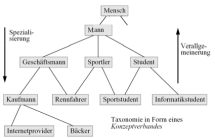
Ausketten

Wert des (ehemaligen) vordersten Element zurückliefern

So kann eine *verkettete Liste* erstellt werden, aus der man auch einen Stack konstruieren kann.

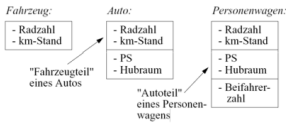
6 Objektorientierung

Die *Objektorientierung* ermöglicht eine Strukturierung in Klassen, wobei eine *Hierarchie* festgelegt wird. Ähnliche Objekte werden zusammengefasst und gemeinsamer Aspekte herausfaktorisiert. Die Klassen beinhalten gewisse Dienstleistungen (Methoden). Objekte, die aus Klassen instanziiert werden, sind autonome, gekapselte Einheiten eines bestimmten Typs und haben eigene Zustände (lokale Variablen). Objekte besitzen ein Verhalten und bieten anderen Objekten Methoden an.



Vererbung: Alle Merkmale des umfassenderen Begriffs werden auf den Unterbegriff vererbt. Somit ist die abgeleitete Klasse immer eine Stufe spezialisierter, die Basisklasse eine Stufe allgemeiner. Somit kann ein Rennfahrer beispielsweise auch als Sportler oder Mensch betrachtet werden (transitiv, is-a-Relation).

Abgeleitete Klassen: Diese besitzt automatisch alle Eigenschaften ihrer Basisklasse (Attribute und Methoden), ausser es werden explizit einige davon unsichtbar gemacht oder redefiniert. Zusätzlich kann die abgeleitete Klasse zusätzliche Attribute und Methoden definieren.

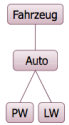



```
class Fahrzeug{
  public int Radzahl;
}

class Auto extends Fahrzeug {
  public int PS;
  public float Hubraum;
}

class PW extends Auto {
  public int Beifahrerzahl;

  void print() {
    System.out.println("Radzahl: " + Radzahl +
                       ", Beifahrerzahl: " +
                       Beifahrerzahl);
  }
}

class LW extends Auto {
  public float Zuladung;
}
```

Vorfahre weiss nicht von den Erben; Erbe muss sich seinen Vorfahren aussuchen!

Erweiterung der Klasse "Fahrzeug": Alles, was in "Fahrzeug" deklariert ist, gehört damit auch zur "Auto" (sowohl Attribute als auch Methoden) – mit gewissen Einschränkungen (→ später)

Auf "weiter oben" definierte Attribute kann ohne weiteres zugegriffen werden – diese sind Teil der abgeleiteten Klasse!

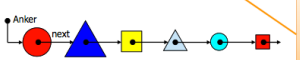


Zuweisungskompatibilität: Objekte von abgeleiteten Klassen können an Variablen vom Typ der Basisklasse zugewiesen werden (Auto ist ein Fahrzeug): `Fahrzeug f; Auto a; f = a;` Die Umkehrung gilt jedoch nicht: `a = f;` ist verboten (Fahrzeug ist nicht immer Auto). Somit können Variablen vom Typ Basisklasse auch ein Objekt der abgeleiteten Klasse enthalten (*Polymorphie*). Somit kann über `f` nicht auf Eigenschaften von `a` zugegriffen werden. Durch Typenkonversion kann dies umgangen werden: `System.out.println(((Auto)f.Hubraum);`

Abstrakte Methoden und Klassen: Werden von den abgeleiteten Klassen jeweils spezifisch implementiert.

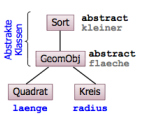
```
abstract class GeomObj {
  GeomObj next;
  public abstract double flaechenwert();}
}
```

Diese Methode muss für jede abgeleitete Klasse mit sinnvoller Semantik versehen werden!



Late binding: es wird zur Laufzeit berechnet, welche konkrete Methode „angesprungen" wird. Ein Anwendungsbeispiel:

```
abstract class GeomObj extends Sort {
  public abstract double flaeche();
  boolean kleiner (Sort y) {
    GeomObj x = (GeomObj)y;
    return flaeche() < x.flaeche();}
}

class Quadrat extends GeomObj {
  double laenge;
  public double flaeche() {
    return (laenge*laenge);
  }
  Quadrat() {
    laenge = KbdInput.readInt("Länge? ");
  }
}

class Kreis extends GeomObj {
  double radius;
  public double flaeche() {
    return (3.1415926536*radius*radius);
  }
  Kreis() {
    radius = KbdInput.readInt("Radius? ");
```



Die Fläche wird jeweils typspezifisch berechnet

Mehrfachvererbung: In Java werden können Klassen nicht von mehr als einer einzigen Basisklasse erben. Einen Teilersatz dafür bieten die Interfaces.

Interfaces: Diese „Schnittstellen" sind rein-abstrakte Klassen, in der alle Methoden deklariert, aber nicht implementiert sind. Ein Interface muss von anderen Klassen implementiert werden:

```
interface Menge {
  int cardinal();
  void insert (Object x);
  void remove (Object x);
}
class S implements Menge {
  public int cardinal(); {
    ...
    while ... i++ ...;
    return i;
  }
```

Alle Methoden sind automatisch public abstract

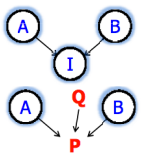
Alle „Variablen" sind automatisch public static final (d.h. „Konstanten")

implements hat die gleiche Rolle wie extends bei „echten" Klassen

Es kann auch eine entsprechende Hierarchie entstehen

Interfaces dürfen mehrere andere Interfaces erweitern, eine Klasse jedoch kann nur eine einzige erweitern, jedoch mehrere Interfaces implementieren:

```
interface A { ...}
interface B { ...}
interface I extends A, B { ...
   int m(); ...
}

class P extends Q
        implements A,B,...
```



7 Exceptions

Die sog. Ausnahmeereignisse sind Fehler-ereignisse, die oft vom System selbst ausgelöst werden. Sie können aber auch explizit im Programm ausgelöst werden. Sie sollten abgefangen und behandelt werden.

```
try {
  value = value / x;
}
catch (ArithmeticException e){
  System.out.println
  ("Division durch 0?");
}
```

Fehlerarten: Ausnahmen können in verschiedensten Situationen auftreten: Ein-/Ausgabe, Netz usw. Eine wichtige Fehlerklasse sind die Laufzeitfehler, wie z.B. ein Zugriff über die `null`-Referenz. Sie können, müssen aber nicht abgefangen werden. Alle übrigen Fehler müssen von einer Methode selbst abgefangen oder explizit weitergeleitet werden. Entweder mit `try` / `catch` oder mit `throws`:

```
import java.io.*;
public eine_methode (...) throws java.io.IOException {
  ... read ...
}
```

Der Aufrufer muss dann mit dieser Exception „rechnen"

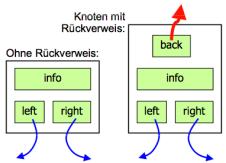
Definieren und Auslösen eigener Ausnahmen: Ausnahmen sind Objekte: es können auch eigene erstellt werden, die dann aber von `java.lang.Throwable` abgeleitet werden müssen.

```
class IllegalesDatum extends Throwable {
  IllegalesDatum(int Tag, int Monat, int Jahr) {
    super("Fehlerhaftes Datum ist: "
          Tag + "." + Monat + "." + Jahr");
  }
}
class Datum {
  ...
  void setzen(int T, int M, int J)
                        throws IllegalesDatum {
    Tag = T; Monat = M; Jahr = J;
    if (Tag > 31)
      throw new IllegalesDatum(Tag, Monat, Jahr);
  }
```

Der Konstruktor von Throwable erwartet einen String, der als Fehlermeldung (mit dem Stack-Trace) ausgegeben wird

8 Binärbäume als Zeigergeflechte

Binärbäume als Referenzstruktur: Es können dynamisch neue Knoten hinzugefügt werden, im Gegensatz zur Array-Repräsentation. Verschiedene Möglichkeiten:




```
class Person {
  String name;
  Person left, right;
  Person back;
}
```

```
int height() {
  if (left!=null && right!=null)
    return 1 + Math.max(left.height(),
                        right.height());
  else if (left!=null)
    return 1+left.height();
  else if (right!=null)
    return 1+right.height();
  else
    return 0;
}
```

Die *Tiefe* eines Knotens ist sein Abstand zur Wurzel. Die *Höhe* eines Baums ist die maximale Tiefe. Ein Baum, der nur aus einer Wurzel besteht, hat also die Höhe 0.

Binäre Suchbäume: Jeder Knoten hat ein Schlüsselattribut und die Menge ist total geordnet. D.h. für jeden Knoten mit Schlüsselattribut s gilt: Alle Schlüssel im linken Unterbaum sind $< s$. Alle Schlüssel im rechten Unterbaum sind $> s$. Das Suchen eines Elements wird somit sehr effizient. Bei gut gefüllten Bäumen ist kommt man schon nach $\log_2(n)$ Schritten an ein Blatt. Wurde das Element bis dahin nicht gefunden, ist es nicht im Baum.

Einfügen in Suchbäume:

```
static void insert (String n,int t,Person p) {
  if (n.compareTo(p.name) < 0)
    if (p.left != null)
      insert(n, t, p.left);
    else {
      p.left = new Person();
      p.left.name = n; p.left.telnr = t;
    }
  else
    if (p.right != null)
      insert(n, t, p.right);
    else {
      p.right = new Person();
      p.right.name = n; p.right.telnr = t;
    }
}
```

```
Person back;
String name;
int telnr;
}
```

Ein neuer Knoten wird immer als Blatt eingefügt

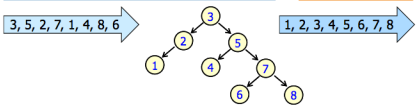
Einfügen der back-Referenz auf den Vorgänger als Denkübung

Inorder-Traversierung:

→ die Ausgabe ist aufsteigend sortiert!

```
static void inorder(Person p) {
  if (p != null) {
    inorder(p.left);
    System.out.println(p.name);
    //System.out.println(p.telnr);
    inorder(p.right);
  }
}
```

→Dirk, Eva, Günter, Jan, Katja, Kerstin, Stefanie, Stephan, Tanja



9 Binärsuche

Binärsuche auf Arrays: Man will feststellen, ob ein Element in einem sortierten Array vorkommt. Dazu prüfen wir, in welcher Hälfte der gesuchte Wert liegen muss. Das ganze dann rekursiv auf die jeweilige Hälfte neu anwenden. Die Suchlänge ist logarithmisch zur Array-Länge: $O(\log_2 n)$.

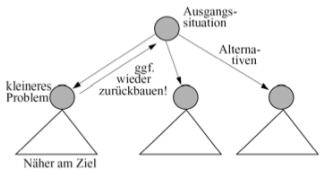
```
import ...
public class BinarySearch {
  public static int binarySearch(Comparable [] a, Comparable x)
                        throws ItemNotFound {
    return binSearch(a, x, 0, a.length -1);
  }

  // Verborgene rekursive Methode:
  private static int binSearch(Comparable [] a, Comparable x,
                               int low, int high)
                               throws ItemNotFound {
    if ( low > high )
      throw new ItemNotFound("BinarySearch fails");
    int mid = (low + high) / 2;
    if ( a[ mid] .compareTo(x) < 0 )
      return binSearch(a, x, mid+1, high);
    else if (a[ mid] .compareTo(x) > 0)
      return binSearch(a, x, low, mid-1);
    else return mid;
  }
```

„Comparable" sei ein Interface mit der Methode „compareTo"

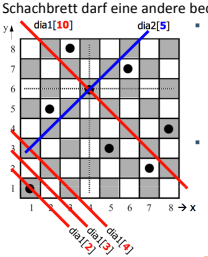
Aber lohnt sich Rekursion bei Binärsuche wirklich?

10 Backtracking

Durchmustern des Entscheidungsbaums: Der Baum aller möglichen Entscheidungen wird systematisch rekursiv durchlaufen. Jeder Knoten stellt eine Entscheidungssituation dar. Gelegentlich wird ein Baum auch dann gekappt, wenn man zwar nicht sicher ist, aber vermutet, dass sich in ihm keine Lösung befindet. Hierfür verwendet man problembezogene Heuristiken.



Das n-Damen-Problem: Keine der n Damen auf einem $n \times n$ Schachbrett darf eine andere bedrohen.



- Darstellung der Spielsituation durch ein (globales) int-Array dame[0..n]:

x	1	2	3	4	5	6	7	8	← Spalte x
dame[x]	1	5	8	6	3	7	2	4	← y-Koordinate

Man braucht also gar kein (aufwendiges) 2-dimensionales Array für den Spielzustand!

- Zweckmässig sind ferner 3 (globale!) boolean-Arrays als „abgeleitete Grössen" aus der Spielsituation:
  - zeile[y] == true: Zeile y ist bedroht
  - dia1[k] == true: Hauptdiagonale mit x+y=k ist bedroht (k=2,...,16)
  - dia2[k] == true: Nebendiagonale mit x-y+7=k ist bedroht (k=0,...,14)

```
static void test (int x) {
  for (int y=1; y<=8; y++)
    if (!(zeile[ y] || dia1[ x+y] || dia2[ x-y+7] )) {
      dame[ x] = y;
      zeile[ y] = true;
      dia1[ x+y] = true;
      dia2[ x-y+7] = true;
      if (x<8)
        test(x+1);
      else {
        for (int y=1; y<=8; y++)
          System.out.print(dame[ i] );
        System.out.println();
      }
      zeile[ y] = true;
      dia1[ x+y] = true;
      dia2[ x-y+7] = true;
    }
}
```

Prüfe, ob Position (x,y) bedroht ist

Hier wird der neue Zustand global gesetzt

rekursiver Aufruf!

Ausgabe einer Lösung

Hier wird der alte Zustand restauriert (unnötig für das Array "dame")

Ansatz: Setze eine (einzige) Dame in Spalte x. Löse dann das Problem rekursiv für das Brett aus den Spalten x+1,..., n. Dabei die Bedrohungen der Damen aus Spalten 1,...,x berücksichtigen!

Beachte: y ist eine lokale Variable jeder Methodeninstanz; die Zustandsvariablen zeile, dia1 und dia2 seien global (in der Klasse) definiert.

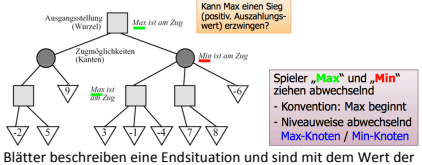
11 Spielbäume

Spieltheorie: Untersuchung des rationalen Verhaltens bei Konflikt- und Konkurrenzsituationen mehrerer Parteien. Jeder Spieler verhält sich dabei rational und ist bestrebt, seinen Gewinn zu maximieren.

Endliche rein strategische 2-Personen-Spiele: Bei endlichen Spiele ist vor allem das Ergebnis interessant. Die 2 Spieler treffen alle Entscheidungen selbst, wobei der Zufall keine Rolle spielt.

Nullsummenspiel / vollständige Information: So hoch wie ein Spieler gewinnt, verlieren die anderen. Es entsteht keine Win-Win-Situation. Der eigene Vorteil ist somit Nachteil des Gegners. Mit den Endsituationen ist für jeden Spieler eine Auszahlungsfunktion definiert. Alle Spieler wissen gleich viel und haben dieselben Informationen, keiner hat einen verdeckten Informationsvorsprung. Somit kann sich jeder Spieler in die Rolle des anderen versetzen.

Spielbäume:



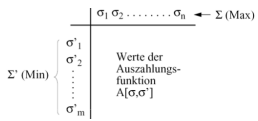
Blätter beschreiben eine Endsituation und sind mit dem Wert der Auszahlungsfunktion für Max markiert. Das Spiel ist durch den zugehörigen Spielbaum vollständig beschrieben. Jeder Ast entspricht einem möglichen Spielverlauf.

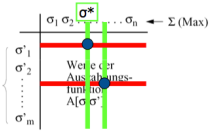
Strategiewahl: Eine Strategie ist ein vollständiger Verhaltensplan, welche für jede Situation eine Handlungsvorschrift aufzeigt. Ein Spiel lässt sich mathematisch daher so auffassen:

$\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ ist die Strategiemenge für Max
$\Sigma' = \{\sigma'_1, \sigma'_2, \ldots, \sigma'_n\}$ ist die Strategiemenge für Min

Bei gewählter Strategie, ist der Spielverlauf vollständig determiniert.

Auszahlungsmatrix: An der Stelle $A[\sigma, \sigma']$ ist der jeweilige Gewinn für den eindeutigen Spielverlauf gespeichert.

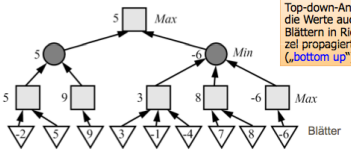




Maximierung des Minimalgewinns: Annahme: optimale Spielweise, vorsichtig und risikolos. Alle Strategiepaare werden betrachtet. Einer festen Gegenstrategie wird die Strategie entgegengesetzt, die am meisten Gewinn einbringt. Der *garantierte Mindestgewinn* G für Max ist $\max_i \min_j A[\sigma_i, \sigma'_j]$ und für Min: $\min_j \max_i A[\sigma_i, \sigma'_j]$. Die zugehörigen Strategien σ^*, σ'^* heissen *optimale Strategien.* Das Paar bildet den Gleichgewichtspunkt. Sie wirken stabilisierend, da für keinen Spieler die Veranlassung besteht, davon abzuweichen.

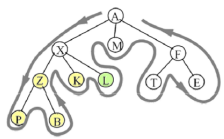
Minimax-Algorithmus: Die optimale Strategie kann effizienter gefunden werden: Sei γ die Auszahlfunktion für Blätter, dann definiere den Minimaxwert $v(k)$ eine Knotens k so:

$$v(k) = \begin{cases} \gamma(k) & \text{, falls k ein Blatt ist} \\ \max\{v(n) \mid \text{n ist direkter Nachfolger von k}\} & \text{, falls k innerer Max-Knoten ist} \\ \min\{v(n) \mid \text{n ist direkter Nachfolger von k}\} & \text{, falls k innerer Min-Knoten ist} \end{cases}$$

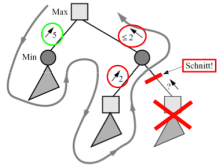


Die Wurzel enthält den Wert, den Max mindestens erreichen kann. Nach der optimalen Strategie wählt Max den grössten direkten Nachfolger, Min den kleinsten.

Tiefensuche: Ein Knoten wird erst erzeugt, wenn alle linken Geschwister und alle deren Nachkommen erzeugt wurden.



Bestensuche: Um schnell zu einem gesuchten Blatt zu gelangen, können Knoten einen Schätzwert erhalten, dann wird unter allen Geschwistern derjenige expandiert, der den besten Schätzwert hat. Man versuchte die Spielbäume möglichst schlank zu halten. Falsche Schätzungen können gute Ergebnisse irrtümlich abschneiden.

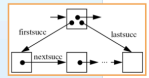


Baumschnitte: Mann kann oft Knoten abschneiden, die den Minimaxwert garantiert nicht beeinflussen. Ziel ist es, solche verzichtbaren Unterbäume möglichst früh abzuschneiden.

Der α-β-Algorithmus: Reduziert den Spielbaum systematisch durch Schnitte, aber liefert den gleichen Minimaxwert der Wurzel wie der eigentliche Minimax-Algorithmus. Beruht auf Tiefensuche, wobei Knoten nur dann expandiert werden, wenn sie den Minimaxwert beeinflussen. Pseudocode:

```
int maxValue(Gamestate g, int α, int β) {
  if cutofftest(g) return eval(g);
  for (GameState s=g.firstsucc; s!=g.lastsucc; s=s.nextsucc) {
    α = max(α, minValue(s, α, β) );
    if (α >= β) break; // β-Schnitt
  }
  return α;
}

int minValue(Gamestate g, int α, int β) {
  if cutofftest(g) return eval(g);
  for (GameState s=g.firstsucc; s!=g.lastsucc; s=s.nextsucc) {
    β = min(β, maxValue(s, α, β) );
    if (β <= α) break; // α-Schnitt
  }
  return β;
}
```

eval ist die statische Bewertungsfunktion für Spielstellungen



Denkübung: ist "value" die richtige Parameterübergabesemantik, oder doch eher "value-result" für α und β?

Definition α- β-Schranken:
α-Schranke: Wert, den Max bei den bisher untersuchten Zügen auf jeden Fall erzielen kann.
β-Schranke: Wert, den Min bei den bisher untersuchten Zügen auf jeden Fall erzielen kann.
α-β-Schranken bilden zusammen ein Suchfenster (α, β).

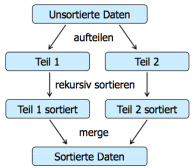
12 Rekursives Problemlösen

Zeitkomplexität des Hanoi-Algorithmus: Die Komplexität ist exponentiell mit $t(n) = 2^n - 1$. Diese Art von Algorithmus wird *inhärent ineffizient* genannt.

```
void hanoi(int groesse,int von,int nach) {
  if (groesse == 1)
    bewege(von, nach)
  else {
    hanoi(groesse-1, von, 6-von-nach);
    bewege(von, nach);
    hanoi(groesse-1, 6-von-nach, nach);
  }
}
```

Divide et impera: 1. Teile die Menge P in 2 Teilmengen P_1, P_2, sodass: $P_1 \cup P_2 = P, P_1 \cap P_2 = \emptyset$. 2. Löse das Problem für die beiden Teilmengen. 3. Vergleiche die Lösung, gebe das Richtige aus (z.B. Minima / Maxima einer Menge bestimmen).

Voraussetzungen: Das Problem muss beim Partitionieren einfacher bzw. kleiner werden. Man muss richtig partitionieren (keine der beiden Teilmengen darf leer werden). Man muss aus den Teillösungen die Gesamtlösung einfach zusammenbauen können.




```
void mergesort(int li, re){
  if (...) {
    m = (li+re)/2;
    mergesort(li,m);
    mergesort(m+1,re);
    ...
```

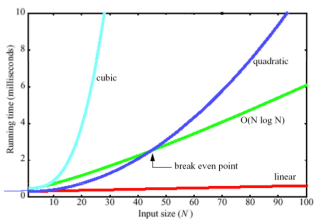
Mergesort (top down rekursiv): Zusammenfassen von 2 sortierten Folgen zu einer einzigen sortierten Folge. 1. Sortiere die erste Hälfte mit Mergesort. 2. Sortiere die zweite Hälfte mit Mergesort. 3. Beide Hälften mergen (Rekursionsabbruch bei Folgengrösse 1). Mergesort benötigt also ca. $n \cdot \log(n)$ Schritte.

13 Komplexität von Algorithmen

Aufwand von Algorithmen: Algorithmen brauchen Rechenzeit und die darin benutzten Datenstrukturen Speicher. Ziel ist es, den Ressourcenverbrauch zu minimieren. Der Problemumfang wird mit n bezeichnet. Der Aufwand wird als $f(n)$ bezeichnet. Beim Zeitaufwand wird von konstanten Faktoren und additiven Termen abstrahiert. Man unterscheidet insbesondere: *best case, average case* und *worst case*. Die Komplexität eines Problems ist der geringstmögliche Aufwand, der mit dem dafür besten Lösungsalgorithmus erreicht werden kann.

Komplexitätsgrössenordnung: Zweck ist die Angabe der Grössenordnung der (Zeit-)Komplexität eines Algorithmus als Funktion der Eingabegrösse n.

Logarithmische Komplexität	$O(\log n)$	z.B. Binärsuche
Lineare Komplexität	$O(n)$	
Quadratische Komplexität	$O(n^2)$	z.B. insertion sort
Exponentielle Komplexität	$O(c^n)$	z.B. Hanoi

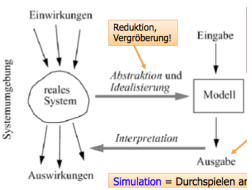


Komplexitätsklassen: Mit $O(f)$ wird jeweils eine ganze Funktionsklasse bezeichnet. $g = O(f)$ heisst somit eigentlich $g \in O(f)$. $g = O(1)$ heisst: g ist beschränkt und überschreitet einen bestimmten Wert nicht. Wichtig ist auch die Klasse der polynomiellen Funktionen: $POLY(n) = \bigcup_{p>0} O(n^p)$.

Komplexität von Sortieren: Das Sortieren von n Elementen ist ein Problem mit Zeitkomplexität $O(n \cdot \log n)$. Mergesort ist grössenordnungsmässig also ein optimaler Algorithmus für das Sortierproblem. Es gibt also keinen „wesentlich" Besseren.

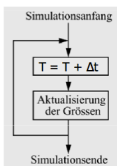
14 Simulation

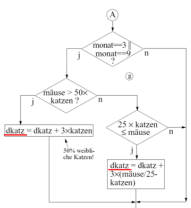
Modelle und Modellierung: Experimente werden anhand eines Modells berechnet, um Rückschlüsse auf die Realität zu ziehen. Wird an Problemen angewendet, für es keine eindeutige Lösung gibt. Simulationen sind im Allgemeinen sehr aufwendig (viel Rechenzeit für Statistik und Detailierungsgrad).



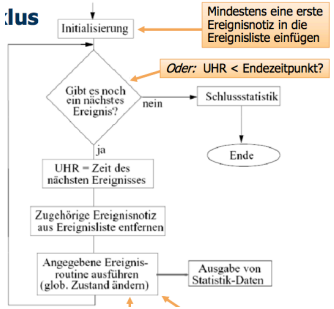
Das Modell ist ein vereinfachtes Abbild der komplexen Realität. Abstraktion ist wichtig, um das Problem lösbar zu machen. Eine zu grobe Abstraktion, kann die Lösung aber verfälschen. Zwecke sind z.B. Optimierungen, Prognosen, Animationen oder Theorienbildungen. Angewendet wird es, weil eine direkte Anwendung in der Realität nicht möglich ist.

Zeitgesteuerte, synchrone Simulation: Zum simulierten Modell gehört auch eine Simulationsuhr, die die aktuelle Zeit des Modells angibt. Pro Simulationsschritt wird die Uhr um eine festes Δt erhöht. Ist Δt zu klein, geht die Simulation zu langsam voran, umgekehrt wird die Simulation zu ungenau, was zu einem Diskretisierungsproblem wird. Was real in $[T, t + \Delta t]$ geschieht, wird erst am Ende der „Epoche", zum Zeitpunkt $T, t + \Delta t$, in der Simulation wirksam. D.h., dass Zustandsänderungen frühestens in der nächsten Epoche Auswirkung zeigt. Zur Planung solcher Auswirkungen und Regelungen helfen solche Flussdiagramme.



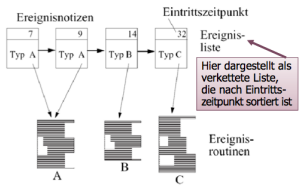


Ereignisgesteuerte Simulation: Die Grundannahme ist, dass Zustände abschnittsweise konstant bleiben, d.h. zwischen 2 Ereignissen passiert nichts. Die Simulationszeit springt also von Ereignis zu Ereignis. Jedes Ereignis hat einen Eintrittszeitpunkt und bewirkt schlagartig eine Zustandsänderung. Es können wieder Flussdiagramme zur Veranschaulichung erstellt werden.



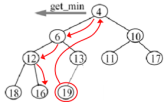
Quasi-Parallelismus: In der Realität gleichzeitig ablaufende Aktivitäten werden in der Simulation „stückweise" sequentialisiert. Durch die Auflösung in Ereignisse werden Aktivitäten zeitlich verzahnt. Die *Modellierungskunst* besteht darin, die Aktivitäten der Realität so in Ereignisse aufzulösen, dass die Wechselwirkung zwischen den Aktivitäten auf die Ereignisse beschränkt bleiben, die Ereignisse sich korrekt gegenseitig einplanen, die zugehörigen Ereignisroutinen die Zustandsänderung des Modells korrekt wiedergeben und das Gesamtverhalten die Realität adäquat widerspiegelt.

Ereignisliste als abstrakter Datentyp: Die Ereignisliste muss nicht als verkettete Liste realisiert sein. *Insert* fügt neue Ereignisnotizen hinzu, *get_min* soll die Ereignisnotiz mit kleinstem Zeitstempel aus der Liste entfernen. Eine Datenstruktur mit diesen beiden Operationen heisst *„priority queue"*. Die *Heap*-Datenstruktur benötigt für beide Operationen nur $O(\log n)$.



15 Heaps

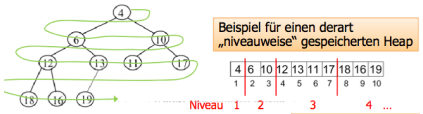
Heap ist ein ordentlicher Binärbaum, wo alle Niveaus ausgefüllt sind, mit Ausnahme des letzten. Für alle Knoten $k \neq$ Wurzel gilt, dass der Wert des Vorgängers kleiner ist als der Wert von k.



Implementierung: *get-min:* 1. Wurzel entfernen. 2. Letzten Knoten des untersten Niveaus an die Wurzelposition setzen. 3. Neue Wurzel so weit wie möglich nach unten sinken lassen: Mit kleinerem der beiden Nachfolger vertauschen.

insert: 1. Als neues Blatt auf unterstem Niveau einfügen. 2. Soweit wie möglich nach oben wandern lassen. Iterativ mit Vorgänger vertauschen, falls dieser grösser.

Den Heap kann man als Array speichern. Nachfolger $2k, 2k+1$.



Als neues Blatt auf unterstem Niveau einfügen.
Soweit wie möglich nach oben wandern lassen: Iterativ mit Vorgänger vertauschen, wenn dieser grösser.

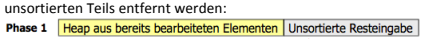
```
void insert (int x) {
  int k = ++N;
  a[ 0] = -1;
  while (a[ k/2] ) >= x) {
    a[ k] = a[ k/2] ;
    k = k/2;
  }
  a[ k] = x;
}
```

N bezeichnet die Anzahl der gespeicherten Elemente
Nur wenn x≥0: Übler "Trick", damit Schleife *immer* verlassen wird
Wird evtl. abgerundet
Pfad hochwandern...

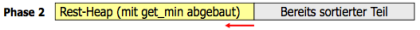
```
int get_min() {
  int k = 1; int j; int x = a[ 1] ;
  a[ 1] = a[ N--] ;
  int w = a[ 1] ;
  while (k <= N/2) {
    j = k + k;
    if (j<N && a[ j+1] < a[ j] )
      j++;
    if (w <= a[ j] )
      break;
    a[ k] = a[ j] ;
    k = j;
  }
  a[ k] = w;
  return x;
}
```

Letzter Knoten wird neue Wurzel
Wurzel nach unten sinken lassen
Platz gefunden: kleiner als beide Nachfolger (falls vorhanden)

Heapsort: 1. n Elemente (der unsortierten Folge) einzeln einfügen. 2. n mal *get_min* auf den Heap anwenden. Dies ist ein Sortierverfahren mit $O(n \log n)$ Worst-Case-Zeitkomplexität. Man kann den Heap im Array selbst aufbauen, indem der Heap von links heranwächst, während nacheinander Elemente des unsortierten Teils entfernt werden:

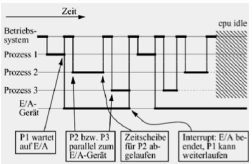


Anschliessend wird der Heap schrittweise abgebaut und das jeweils entfernte Element an eine im freigeräumten Bereich heranwachsende sortierte Folge angefügt:

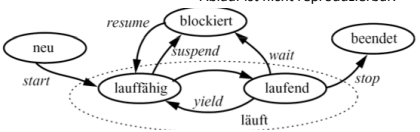


16 Parallele Prozesse und Threads

Prozesse und Betriebsmittel: Es kann gleichzeitig mehrere Prozesse (Programm in Ausführung) als verschiedene Instanzen des gleichen Programms geben. Der Kontext eines Prozesses umfasst: aktuelle Stelle, Inhalt der CPU-Register, Werte aller Variablen, Inhalt des Laufzeitstacks, Zustand zugeordneter Betriebsmittel. Ein Prozess benötigt Betriebsmittel („Ressourcen"). Sie werden durch das Betriebssystem verwaltet (Terminieren, Ressourcenverbrauch, usw.).



Multitasking: quasi-gleichzeitiges Ausführen mehrerer Prozesse. Das Multiplexen der CPU entspricht einem time-sharing der einzelnen Prozesse. Somit laufen die Prozesse eigentlich sequentiell, werden aber als gleichzeitig wahrgenommen. Der Ablauf ist nicht reproduzierbar.



Prozesszustände: Ein Prozess kann entweder blockiert sein, oder laufen. Zu einem Zeitpunkt ist stets nur ein einziger Prozess tatsächlich laufend, die lauffähigen warten darauf, ein wenig CPU-Zeit zu bekommen. Der Zustandswechsel zwischen lauffähig und laufend wird vom Scheduler vorgenommen.

Prozesskontrollblock: Für jeden Prozess wird ein Kontrollblock angelegt, der alle notwenigen Informationen enthält. Wird der laufende Prozess unterbrochen, muss der aktuelle Kontext des Prozesses gesichert werden (Programmzähler, Inhalt der Register, usw.). Der zu sichernde Prozesszustand ist recht umfangreich. Ein Kontextwechsel ist relativ teuer.

Multithreading: Threads sind parallele Kontrollflüsse, die nicht gegeneinander abgeschottet sind und sich gemeinsame Ressourcen teilen. Ein Kontextwechsel ist hier viel effizienter (kein Adressraumwechsel, usw.). Man kann also quasi-gleichzeitig mehrere Vorgänge innerhalb einer einzigen Anwendung erledigen.

Programmstruktur eines Threads:
Die Klasse `java.lang.Thread`

```
- void start()
- void suspend()
- void stop()
```

```
- void resume()
- void wait()
- void yield()
```

beinhaltet:

```
- void sleep(long millis) // blockiert einige ms
- void join() // Synchronisation zweier Threads
- int getPriority()
- void setPriority(int prio)
- void setDaemon(boolean on)
```

„Hintergrundprozess": terminiert *nicht* mit dem Erzeuger

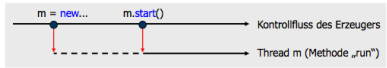
Jeder Thread muss eine `void`-Methode `run()` enthalten: Typisches Gerüst:

```
class MyThread extends Thread{
  int myNumber;
  public MyThread(int number) { // Konstruktor
    myNumber = number;
  }
  public void run() {
    // hier die Anweisungen des Thread
  }
  // hier weitere Methoden
}
```

Erzeugen eines Threads:

```
MyThread m = new MyThread(5);
m.start();
```

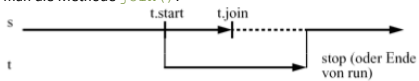
Mit dieser Nummer identifizieren wir einen Thread individuell
Damit kann man auf den Thread zugreifen und diesen kontrollieren (z.B. m.suspend();)



Mit einer anonymen Erzeugung `MyThread(5).start();` gibt man jegliche Kontrolle ab.

Thread-Steuerung: Ein Thread läuft solange, bis die `run`-Methode zu Ende ist oder `stop()` aufgerufen wird. Derweil kann ein Thread *sich selbst:* `yield()` (Übergang von laufend zu lauffähig), `sleep()` (wie suspend, mit resume nach gegebener Zeit), `suspend()`, `stop()`, `setPriority()`. Ein Thread kann einen anderen Thread: `start()`, `suspend()`, `resume()`, `stop()`, `setPriority()`. Man beachte, dass stop, suspend, resume zu unsicheren Programmen führen kann.

Thread-Ende: Das Objekt eines beendeten Threads existiert weiter und kann mit start wieder neu loslaufen. Soll auf die Beendigung eines anderen Threads gewartet werden, verwendet man die Methode `join()`:



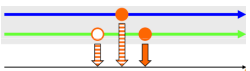
Nach `t.join()` ist in jedem Fall garantiert, dass `t` beendet ist.

Thread-Scheduling: Planvolle Zuordnung der CPU an die einzelnen Threads. Eine genaue Scheduling-Strategie ist in Java nicht standardisiert. Der Scheduler selbst sollte mit höchster Priorität laufen.

Prioritäten: Ein Thread-Scheduler soll Threads mit höherer Priorität bevorzugen. Wird ein höher-prioritärer Thread lauffähig, wird sofort dieser laufend gemacht und der vorige suspended.

Schwierigkeiten: Eine Endlosschleife in einem Thread kann unter Umständen das ganze System blockieren. Bei Prozessoren mit mehreren CPUs könnten entsprechend viele Threads echt gleichzeitig ausgeführt werden. Das Programmieren und Debuggen von Threads ist schwierig.

Race-Conditions: Threads kommunizieren über gemeinsame Variablen. Dabei kann es zu unterschiedlichen Ergebnissen kommen, je nachdem, welcher Thread zuerst oder zuletzt auf eine Variable zugreift.



Relative Atomarität: Atomare Folge von Operationen: Während die Folge ausgeführt wird, werden keine anderen Operationen quasi-gleichzeitig ausgeführt. Wenn die CPU mit der ersten Operation der atomaren Folge beginnt, arbeitet sie diese bis zur letzten ab, ohne zwischendrin etwas anderes zu tun. Unterbrechungen sind höchstens zwischen atomaren Folgen erlaubt. Unkritische Dinge könnten aber parallel zur atomaren Folge ausgeführt werden. Diese Begriffe, wie atomar und unkritisch sind jedoch stark relativ.

Inkonsistenzen: Durch die Nicht-Atomarität von Anweisungsfolgen kommt es bei paralleler Ausführung leicht zu unerwünschten Effekten. Dafür sind Unterbrechungssperren notwendig.

Kritischer Abschnitt: Folge von Anweisungen, die bezüglich anderen entsprechenden kritischen Abschnitten wechselseitig ausgeschlossen ist. D.h. während ein Thread im kritischen Abschnitt ist, darf kein anderer Thread einen entsprechenden kritischen Abschnitt betreten. Höchstens einer hat also die Erlaubnis. In einem kritischen Abschnitt werden diejenigen elementaren Operationen ausgeführt, die ungestört als Ganzes ausgeführt werden müssen. Es gibt 3 Anforderungen an solch einen kritischen Abschnitt: 1. *Safety*: Ein Prozess in kritischem Abschnitt, kein anderer. 2. *Liveness*: Wenn kein Prozess im kritischen Abschnitt ist, aber einige wollen, kommt schliesslich einer in den kritischen Abschnitt. 3. *Fairness*: Ein sich bewerbender Prozess darf nicht beliebig oft von anderen übergangen werden.

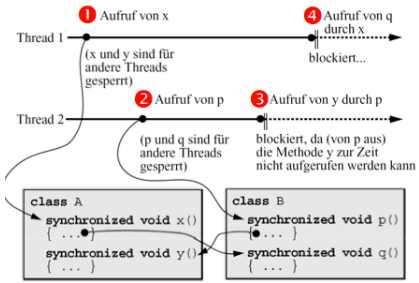
```
synchronized (xxx) {
  // Anweisung 1
     ...
  // Anweisung n
}
```

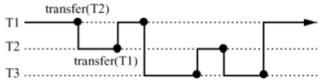
Synchronized: Bei Java wird ein kritischer Abschnitt durch eine in {}-geklammerte Anweisungsfolge mit vorangestelltem Schlüsselwort `synchronized` spezifiziert.

Alle bezüglich des gleichen Sperrobjekts synchronisierten kritischen Abschnitte schliessen sich wechselseitig aus. Es können auch ganze Methoden mit `synchronized` gekennzeichnet werden.

```
synchronized void Update (int betrag) {
  konto = konto + betrag;
  System.out.println(konto);
}
```

Deadlock-Problem: Im Beispiel: Thread 1 wartet auch Freigabe von B durch Thread 2 und Thread 2 wartet auf Freigabe von A durch Thread 1 → Deadlock.





Atomarer Kontrolltransfer: Bei Nutzung von Threads taucht oft folgendes weitere Problem auf: Es soll (abwechselnd) immer nur einer von n Threads laufen. Jeder Thread soll an einer gewissen Stelle die Kontrolle an einen bestimmten anderen Thread transferieren. Wenn die Kontrolle zurücktransferiert wird, dann soll genau an dieser Stelle weitergemacht werden. Lösungen mit `suspend`/`resume` oder mit `synchronized` führen schnell zu Deadlocks. Eine mögliche Lösung ist mit `wait`/`notify` möglich.

Anhang

Java-Bytecode:

iconst_m1
Push the integer -1 onto the stack.

iconst_0, ... iconst_5
Push the integer 0,..., 5 onto the stack.

iload_vindex
The value of the local variable at vindex in the current Java frame is pushed onto the operand stack.

istore_vindex
Local variable vindex in the current Java frame is set to value.

nop
Do nothing

bipush byte
Push one-byte signed integer.

pop
Pop top stack word from the stack.

iadd
Value1 and value2 must be integers. The values are added and replaced on the stack by their integer sum.

imul
...replaced on the stack by their integer product.

vindex: die Variablen eines „frames" sind durchnummeriert; vindex ist dabei die Nummer („index") einer Variablen

Komplexitätsanalyse: Beispiele:

`while(n >= 1) n = n/2;` $\Rightarrow O(\log(n))$

```
for (int i=0; i<n; i++)
    for (int j=0; j<i; j++)
        a++;
```

$$\Rightarrow \sum_{i=0}^{n}\sum_{j=0}^{i} 1 = 1 + 2 + \cdots + (n^2 - 1) + n^2 \quad \Rightarrow \quad O(n^2)$$

Laufzeitanalyse: $O(2^n) \leftrightarrow M_1$ mit gegebener Effizienz. Berechnen der Menge bei 3facher Effizienz:

$$2^{M_1} = T_1 \Rightarrow 2^{M_1'} = 3T_1 \Rightarrow \log_2(3T_1) = M_1'$$

$$\Rightarrow \log_2(3 \cdot 2^{M_1}) = M_1' = \log_2(3) + M_1 \cdot \overbrace{\log_2(2)}^{1} \approx M_1$$