Referenztypen – Teil 1

Inhalt:

1. Referenz- und primitive Datentypen
2. Strings als Referenzen
3. Felder als Referenzen
4. Mehrdimensionale Felder als Referenzen
5. Methoden, Parameter, Blöcke usw.
   5.1 Methoden
   5.2 Parameter und der Stack
   5.3 Rekursionen
   5.4 Blöcke und Namensbereiche
6. Beispiel: Genetische Algorithmen

Bezug:
G. Krüger, GotoJava 2 – HTML-Version:
Kapitel 4.2, 4.4, 4.5
Kapitel 11.3

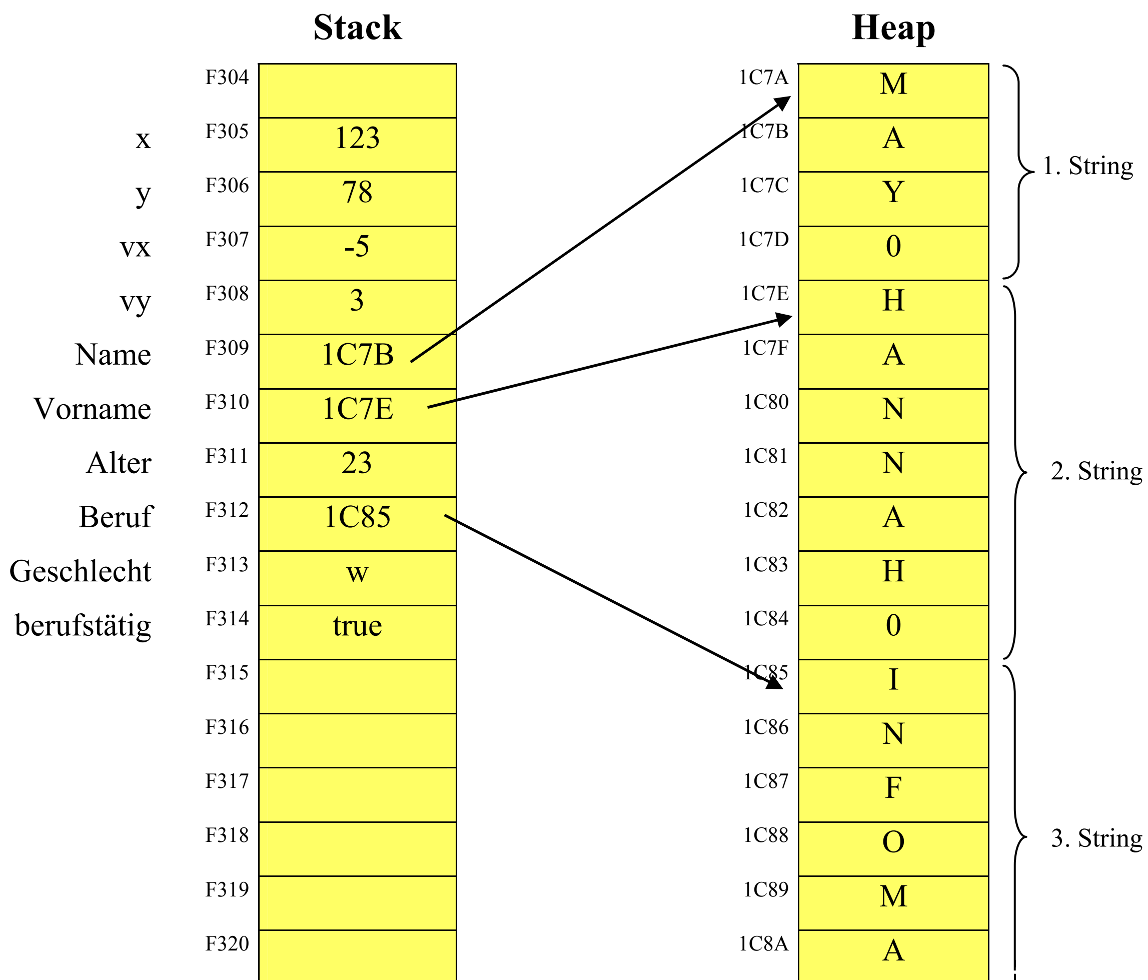
1. Referenz- und primitive Datentypen

Will man Daten im Rechner ablegen („speichern“), dann benötigt man Platz dafür. Um zu vermeiden, sich direkt mit dem Speichermanagement eines Rechners auseinandersetzen zu müssen, werden *symbolische Namen* für *Variable* eingeführt, hinter denen sich eine Speicheradresse im Computer verbirgt. Das Programmentwicklungssystem – z. B. J++ – führt dann eine Art „Adressbuch“, in dem die Zuordnung von Variablennamen zu Speicheradressen vermerkt ist. (Bei der Übersetzung des Programms werden die Namen weitgehend durch Adressen ersetzt.)

Variable werden in einem besonders organisierten Speicherbereich – dem *Stack* („Stapel“) abgelegt, der meist eine ziemlich überschaubare Größe hat. Um den nicht zu überfordern, werden hier nur die „kleinen“ Datentypen direkt gespeichert: die *Simple Types* oder *Primitiven Datentypen*. Dazu gehören Zahlen, boolesche Werte und Zeichen.

Die „großen“ Datentypen (*Referenztypen*) werden anders verwaltet: Auf dem Stack wird nur ein *Zeiger* (eine *Referenz* oder Speicheradresse) abgelegt, der auf einen Speicherbereich an anderer Stelle verweist, die irgendwo im restlichen freien Speicher liegt. Dieser wird als *Heap* (Haufen) organisiert und vor allem für die Speicherung großer Datenmengen genutzt. Von den Standarddatentypen Javas gehören Zeichenketten (Strings), Felder (Arrays) und Objekte zu den Referenztypen.

Veranschaulichen wir die Speicherzellen durch einzelne Kästchen, dann können die Kästchen des Stacks mithilfe ihres Variablennamens erreicht werden, die Werte auf dem Heap aber nur über die entsprechenden Referenzen. ***Heap-Speicherbereiche haben keine Namen.***



Der Speicherplatz auf dem Heap muss vom laufenden Programm bei Bedarf vom Betriebssystem angefordert werden. Das geschieht mit dem *new*-Operator. Die Größe des benötigten Speicherbereichs ergibt sich aus dem Typ, der dem *Konstruktor* (einer besonderen Methode der Klasse, die denselben Namen wie die Klasse selbst haben muss) dieses Typs bekannt ist. Haben wir also eine *Klasse Ball* mit den Datenfeldern *int x,y,vx,vy* vereinbart, dann bewirkt der Aufruf des *Konstruktors Ball* innerhalb von

```
Ball b = new Ball(10,10,-5,3);
```

dass in dem Stackbereich mit dem symbolischen Namen *Ball* die Adresse eines 4*32-Bit großen Speicherbereichs eingetragen wird. Von diesem können die ersten 32 Bit den Wert des Datenfeldes *x*, die nächsten den Wert von *y* usw. aufnehmen. Der Zugang zu diesem Speicherbereich erfolgt über die Referenz namens Ball.

Sind Referenzvariable noch nicht über einen *new*-Aufruf instantiiert worden, dann enthalten sie den Wert *null*, einen mit allen Referenztypen kompatiblen Zeiger „ins Nichts".

Da Referenztypen lediglich Adressen beinhalten, werden bei Zuweisungen und Vergleichen auch nur diese Adressen kopiert bzw. verglichen. Meint man (wie meist) die gespeicherten Inhalte, dann muss man das direkt angeben (s.u.).

Felder und Zeichenketten sind eigentlich normale Objekte. Da sie aber so oft benötigt werden, „weiß" der Compiler über ihren inneren Aufbau mehr als bei anderen Klassen. Es gibt deshalb Möglichkeiten, entsprechende Variable ohne direkten new-Aufruf direkt zu instantiieren (durch Zuweisung von Literalen). Dieses erleichterte Verfahren ist zwar ganz praktisch, führt aber zu Inkonsistenzen in der Benutzung der entsprechenden Typen und erschwert damit das Verständnis.

2. Strings als Referenzen

Ein solches „Verständnisproblem" ergibt sich aus der unterschiedlichen Behandlung von Strings, die mit dem new-Operator bzw. durch Zuweisung von Literalen instantiiert wurden:

- Der Compiler führt eine Liste der im Programm benutzten Literale (fester Zeichenketten). Weist man verschiedenen String-Variablen das gleiche Literal zu, dann verweisen diese wirklich auf die gleiche Adresse, d. h. ***Referenz und Inhalt sind gleich***.
- Erzeugt man String-Referenzen mithilfe des new-Operators, dann erhält man ***verschiedene Referenzen, auch wenn die Inhalte gleich sind***.
- Erzeugt man String-Referenzen unterschiedlichen Inhalts mithilfe des new-Operators und weist der einen anschließend den Wert der anderen zu, dann erhält man ***gleiche Referenzen***.

Zum Test wollen wird die drei Fälle in den Ereignisbehandlungsmethoden dreier Buttons durchspielen:

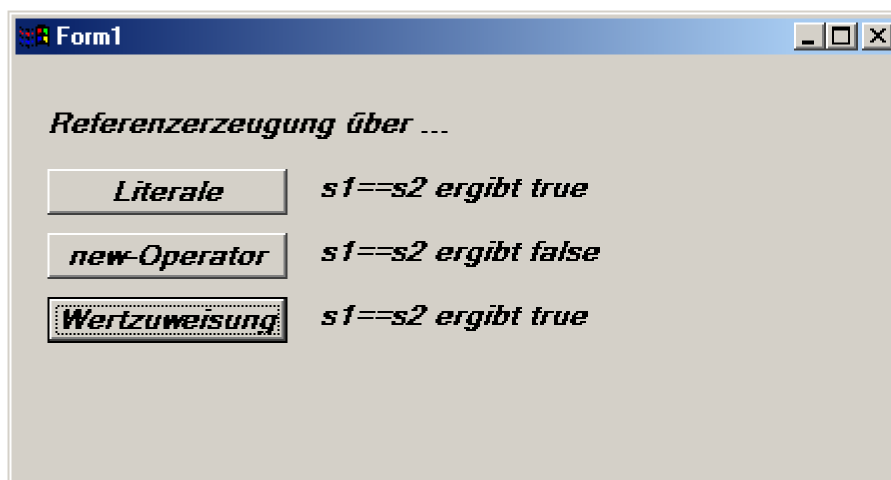
```
private void b_literale_click(Object source, Event e)
{
     String s1="Test",s2="Test";
     label1.setText("s1==s2 ergibt "+(s1==s2));
}


private void b_new_click(Object source, Event e)
```

```
{
     String s1=new String("Test"),s2=new String("Test");
     label2.setText("s1==s2 ergibt "+(s1==s2));
}

private void b_wertzuweisung_click(Object source, Event e)
{
     String s1=new String("Test1"),s2=new String("Test2");
     s1 = s2;
     label3.setText("s1==s2 ergibt "+(s1==s2));

}
```



Echte Inhaltsvergleiche führt man deshalb besser mit der Methode *equals()* aus. Verändern wir das zweite Beispiel entsprechend, dann liefert das folgende Programmstück das erwarte Ergebnis:

```
String s1=new String("Test1"),
       s2=new String("Test2");
label3.setText("s1.equals(s2) ergibt "+s1.equals(s2));
```

3. Felder als Referenzen

Arrays sind in Java ebenfalls Referenzen, d. h. der Speicherbereich, auf den sie verweisen, muss ihnen im Programm erst zugewiesen werden. Das kann entweder mithilfe des new-Operators geschehen oder durch Zuweisung eines Literals. Als Folge sind Arrays *semidynamisch*: vor dieser Initiierung liegt ihre Größe nicht fest, kann also vom Programm bestimmt werden, nach dieser Zuweisung bleibt die Größe unveränderlich.

Wenn wir mit Feldern arbeiten, dann müssen wir zuerst entsprechende Variable deklarieren. Dazu hängen wir an den Datentyp, der im Array gespeichert werden soll, eckige Klammern an:

```
int[] zahlenfeld;
```

```
double[] messwerte, gemittelteWert;
```

Zu einem späteren Zeitpunkt wird das Array instantiiert, indem Platz für eine bestimmte Anzahl von Elementen angefordert wird, die dem angegebenen Feldtyp entsprechen.

```
zahlenfeld = new int[100];     //Platz für 100 ganze Zahlen
messwerte  = new double[20];   //Platz für 20 Gleitpunktzahlen
```

Ggf. kann diese Instantiierung auch zusammen mit der Deklaration erfolgen (wovon ich zumindest anfangs aber abraten möchte):

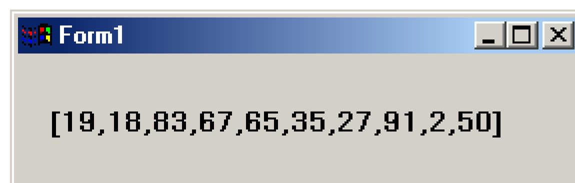
```
int[] zahlenfeld = new int[100];
```

Falls die Inhalte des Arrays schon bei der Instantiierung bekannt sind, können diese auch als Literale angegeben werden. Dazu werden sie in *geschweifte Klammern* gesetzt.

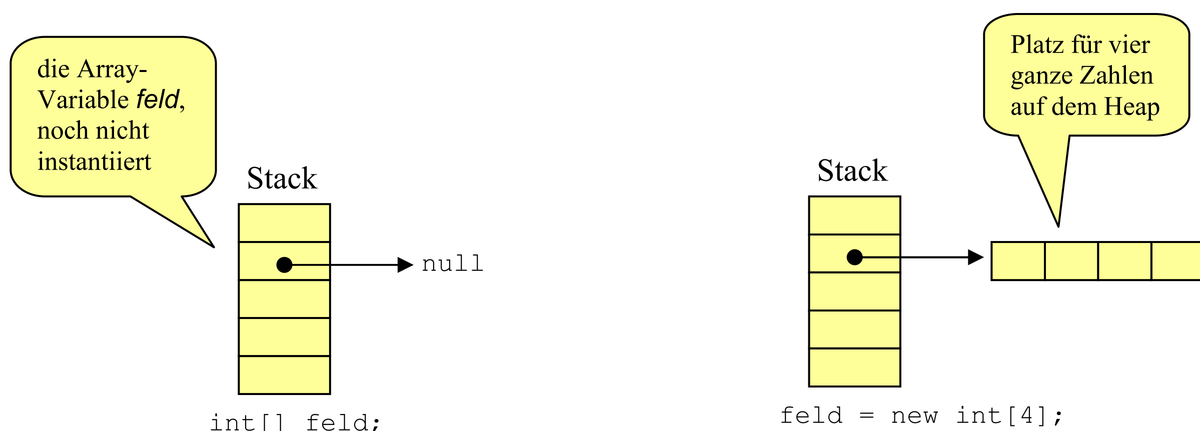
```
int[] zahlenfeld = {1,2,3,4,5,6,7};
```

Sind in einem Feld *n* Elemente gespeichert, dann läuft der Index von *0* bis *n-1*. Die Anzahl der gespeicherten Elemente kann mit der *Instanzvariablen* length (das ist keine Methode!!!) abgefragt werden.

```
int[] zahlenfeld = new int[10];
String h;
for(int i=0;i<zahlenfeld.length;i++)
     zahlenfeld[i] = (int)Math.round(Math.random()*100);
h = "[";
for(int i=0;i<zahlenfeld.length-1;i++)
     h=h+zahlenfeld[i]+",";
h=h+zahlenfeld[zahlenfeld.length-1]+"]";
label1.setText(h);
```



Stellen wir uns den Stack wieder als einen Speicherstapel vor und veranschaulichen den Heap diesmal durch einzelne Speicherzellen, dann erhalten wir für die Funktionsweise von Arrays das folgende Modell:



4. Mehrdimensionale Felder als Referenzen

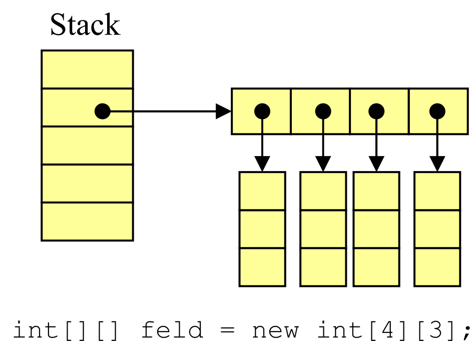
Da die Inhalte von Feldern zwar aus einem, aber einem beliebigen Datentyp bestehen, können Felder auch Felder speichern. Mehrdimensionale Felder werden deshalb als Arrays von Arrays angelegt. (Eine „krasse“ Folge dieser Organisation ist, dass Felder nicht rechteckig zu sein brauchen. Die einzelnen „Unterfelder“ können ruhig unterschiedliche Größen haben.) Ein rechteckiges Feld kann z. B. deklariert werden als:

```
int[][] zahlenfeld = new int[10][5];
```

Auf ein Elemente wird dann zugegriffen z. B. mit

```
zahlenfeld[1][2] = 3;
```

In unserem Modell sähe ein rechteckiges Feld nach der Instantiierung so aus:



`int[][] feld = new int[4][3];`

Rechteckige Felder finden sich in sehr vielen Anwendungsgebieten:

- Grafiken werden als „Bildpunktfelder“ (Bitmaps) gespeichert, deren Feldelemente die Farbinformationen als Zahlenwerte beinhalten.
- Messwerte von Empfängern der astronomischen Großteleskope liegen als „Counts“ ebenfalls in einer Art Bitmap vor (meist mit vielen Zusatzinformationen im FITS-Format).
- Die Resultate von PET-Aufnahmen der Neurologie liegen als ebene Schnitte im Feldformat vor.

Meist können diese Informationen über eine *Falschfarbendarstellung* grafisch aufbereitet werden. Es ergeben sich hübsche Aufgaben.

Ein „dreieckiges“ Array kann z. B. zur Darstellung des *Pascalschen Dreiecks* benutzt werden:

```
public class Applet1 extends Applet
{

  public void paint(Graphics g)
  {
    int[][] pascalDreieck = new int[10][];

    for (int i=0;i<10;i++)
    {
      if (i>0) pascalDreieck[i] = new int[i+2];
      else pascalDreieck[i] = new int[1];
```

mehrdimensionales Feld, in einer Dimension vereinbaren

je nach Index unterschiedliche Unterfelder“ vereinbaren

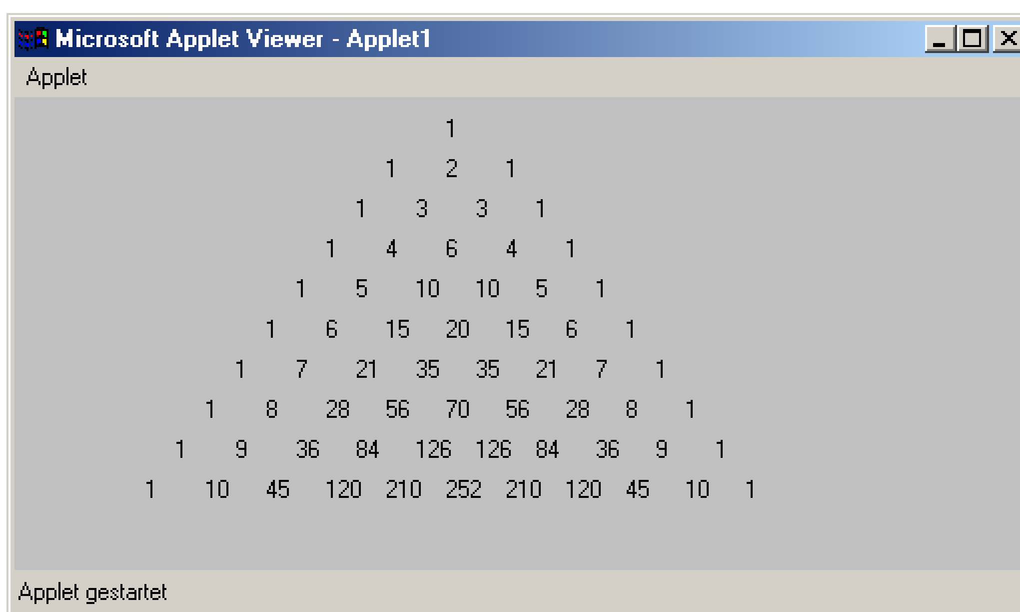
```
      switch(i)
       {
         case 0: {pascalDreieck[0][0]=1; break;}
         case 1: {
                   pascalDreieck[1][0]= 1;
                   pascalDreieck[1][1]= 2;
                   pascalDreieck[1][2]= 1;
                   break;
                 }
         default:
                 {
                   for (int j=0;j<i+1;j++)
                     if (j==0) pascalDreieck[i][0] = 1;
                     else pascalDreieck[i][j] =  pascalDreieck[i-1][j-1]
                                              + pascalDreieck[i-1][j];
                   pascalDreieck[i][i+1]=1;
                 }
       }
     }

     int x,y;
     String h;

     for (int i=0;i<10;i++)
     {
       g.setColor(Color.black);
       for (int j=0;j<pascalDreieck[i].length;j++)
       {
         h = String.valueOf(pascalDreieck[i][j]);
         x = 200-15*i+30*j;
         if (i==0) x=x+15;
         y = 20*i+20;
         g.drawString(h,x,y);
       }
     }
   }
 }
```

Werte eintragen

Pascalsches Dreieck darstellen

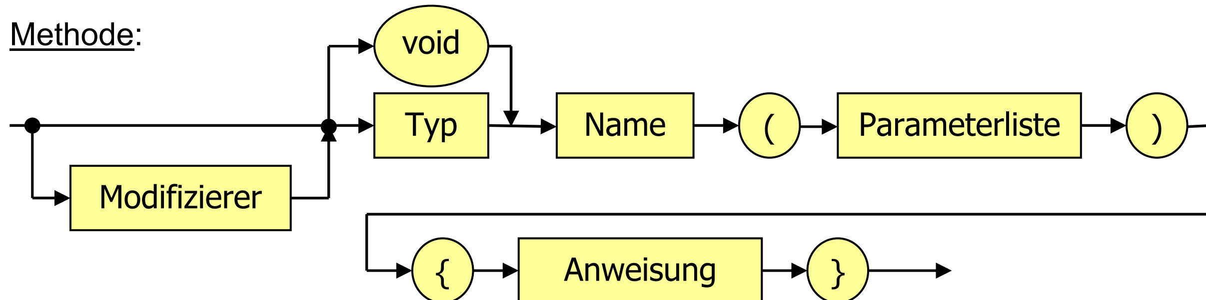


5. Methoden, Parameter, Blöcke usw.

5.1 Methoden

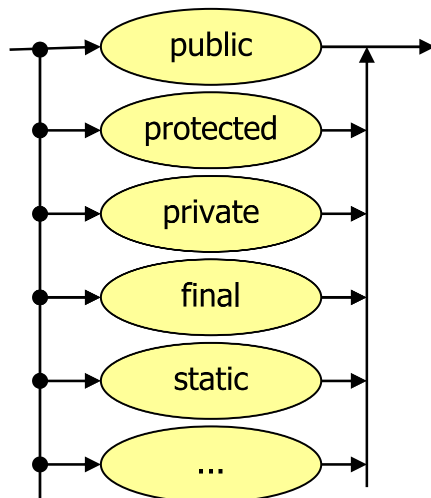
Mit Java kann man keine alleinstehenden Unterprogramme wie in den meisten älteren Programmiersprachen schreiben, sondern man versieht ausschließlich Objekte mit *Methoden*, die die Funktionalität der jeweiligen Klasse implementieren. Die (nicht ganz vollständige) Syntax einer Methode lautet:

Methode:



Modifizierer bestimmen u. a. die *Sichtbarkeit* einer Methode, d. h. sie geben an, wer wann die Methode aufrufen kann. Wir behandeln hier nur einen Teil der möglichen Werte:

Modifizierer:

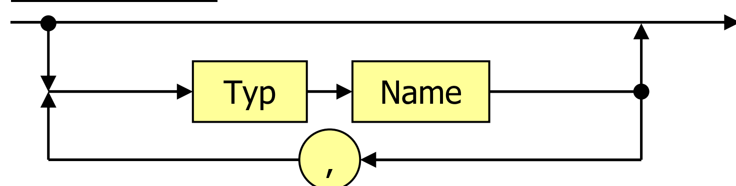


Typ: einer der Java-Standardtypen oder eine Klasse oder *void*, wenn kein Wert zurückgegeben wird

Name: wie üblich

Anweisung: wie üblich

Parameterliste:



Der Modifizierer *public* bewirkt, dass eine Methode nicht nur innerhalb ihrer Klasse und daraus abgeleiteter, sondern auch außerhalb des Pakets benutzt werden kann. Die Methode ist also „frei verfügbar". Wird eine Klasse anderen Programmen zur Verfügung gestellt, dann muss mindestens eine Methode als *public* deklariert werden (sonst „kommt man gar nicht ran").

Der Modifizierer *protected* schützt eine Methode vor unberechtigtem Zugriff aus „fremden" Klassen. Innerhalb der Klasse, aus abgeleiteten Klassen und innerhalb des Pakets ist sie zugänglich.

Der Modifizierer *private* schützt eine Methode umfassend vor unberechtigtem Zugriff. Sie kann nur innerhalb der Klasse benutzt werden. In abgeleiteten Klassen ist sie ebenfalls unbekannt.

Der Modifizierer *final* definiert eigentlich Konstante. Bei Methoden bewirkt er, dass diese nicht überlagert werden können.

Der Modifizierer *static* bewirkt, dass eine Methode aufgerufen werden kann, ohne dass ein Objekt dieses Typs instantiiert wird. *Statische Methoden* stellen also eine Funktionalität bereit, die unabhängig von Instanzen abgerufen werden kann. Typische Vertreter sind die statischen Methoden der Sting-Klasse, mit denen z. B. primitive Typen in Strings verwandelt werden können (Bsp: `String.valueOf(123)`).

Kommen wir jetzt zum *Typ* einer Methode: In Java werden (wie in anderen Sprachen mit C-ähnlicher Syntax) Methoden generell wie Funktionen anderer Sprachen gehandhabt. Sie geben also „normalerweise" einen Wert zurück, dessen Typ vor dem Methodennamen angegeben wird. Der „Ausnahmefall", dass die Methode nur aufgerufen wird, nur um etwas zu tun, also keinen Wert zurückgibt, wird durch das Schlüsselwort *void* gekennzeichnet. (*void*-Methoden entsprechen damit den *Prozeduren* anderer Sprachen).

`private int test(){…}` liefert eine ganze Zahl zurück, ist also eine Funktion

`private void test(){…}` liefert nichts zurück, ist also eine Prozedur

Die Klammern nach dem Methodennamen kennzeichnen eine Methode. Sie sind unbedingt erforderlich, auch dann, wenn keine Parameter übergeben werden. Die beiden Größen `public int test` und `public int test()` stellen deshalb unterschiedliche Dinge dar (eine Variable und eine Methode).

5.2 Parameter und der Stack

Die Parameterliste von Methoden kann also leer sein. Enthält sie Werte, dann werden die ähnlich wie Variable deklariert, aber immer einzeln, durch Kommas getrennt. Gültige Parameterlisten (mit dazugefügten Klammern) sind:

```
(int i, int j)      (String Name, int Alter, double Gehalt)
```

Methoden werden über ihren Namen *aufgerufen*. Verfügen sie über eine Parameterliste, dann müssen für alle dort angegebenen Parameter Ausdrücke des richtigen Typs stehen, so dass beim Aufruf die so berechneten Werte an die Parameter übergeben werden können. Als Beispiel wählen wir eine Funktion und eine Prozedur:

```
public int berechne(int x)
{ return x+5; }

public void teste()
{ int y; y = berechne(2+3); }
```

Beim Aufruf wird der Wert *5* ermittelt und dem Parameter *x* zugewiesen. Die Variable *y* erhält den zurückgegebenen Wert *10*.

Man kann auch Methoden mit gleichem Namen (und dann sinnvoller Weise auch ähnlicher Funktionalität), aber mit unterschiedlichen Parameterlisten definieren. Anhand der Parameterliste wird dann die „richtige" der so ***überladenen*** Methode ausgewählt.

Parameter werden in Java „eigentlich" als *Werteparameter* übergeben (*call by value*). Für primitive Datentypen gilt das auch uneingeschränkt. Die Namen in der Parameterliste sind rein *symbolische* Namen. Sie stellen keinerlei Verbindung zu den Variablen oder Werten dar, mit denen die Methode aufgerufen wurde. **Parameter werden also nur über ihre Reihenfolge in der Parameterliste identifiziert.** Eine Bedeutung haben sie nur innerhalb der Methode. Dort gehören sie zum gleichen *Namensraum* wie die anderen dort vereinbarten *lokalen* Größen. Findet man in der Parameterliste den Namen *a*, dann ist dieser in der Methode vergeben. Eine weitere Variable dieses Namens darf dort nicht vereinbart werden.

Werden Referenztypen als Parameter benutzt, dann hat die einfache Welt der Werteparameter ihre Grenzen erreicht. Auch hier werden zwar nur Werte übergeben. Da diese aber Referenzen

darstellen, verweisen die Parameter auf die gleichen Speicherbereiche wie die Referenzen außerhalb der Methode. Solche Parameter verhalten sich also wie *Referenzparameter* anderer Sprachen. Sie gestatten Änderungen außerhalb der Methode (*call by reference*). Beachtet man das nicht, dann können böse Fehler die Folge sein.

Parameter und lokale Variable werden auf dem Stack abgelegt, und zwar in dem freien Bereich ab und oberhalb eines Zeigers, der den ersten freien Platz auf dem Stack markiert: dem *Stackpointer*. Da das Programm „weitermachen“ muss, nachdem eine Methode aufgerufen und abgearbeitet worden ist, „merkt“ sich das Programm auf dem Stack auch die *Rücksprungadresse*, also den ersten Befehl, der nach Beendigung der Methode ausgeführt werden soll. (Dort „geht es weiter“.) Bei Funktionen wird zusätzlich Platz für den Rückgabewert gelassen. Die für den aktuellen Aufruf einer Methode auf dem Stack abgelegten Parameter und Variablen enthalten den aktuellen Satz von Werten, mit denen die Methode gerade arbeitet. Wird eine Methode mehrfach aufgerufen, dann wird bei jedem Aufruf ein vollständiger Satz dieser Werte neu abgelegt. **Unterschiedliche Methodenaufrufe arbeiten deshalb mit unterschiedlichen Parameter- und Variablenwerten!** Wir wollen das an einem einfachen Beispiel veranschaulichen: Gegeben sei die Methode

```
private int summe(int a, int b)
{ int s=a+b; return s; }
```

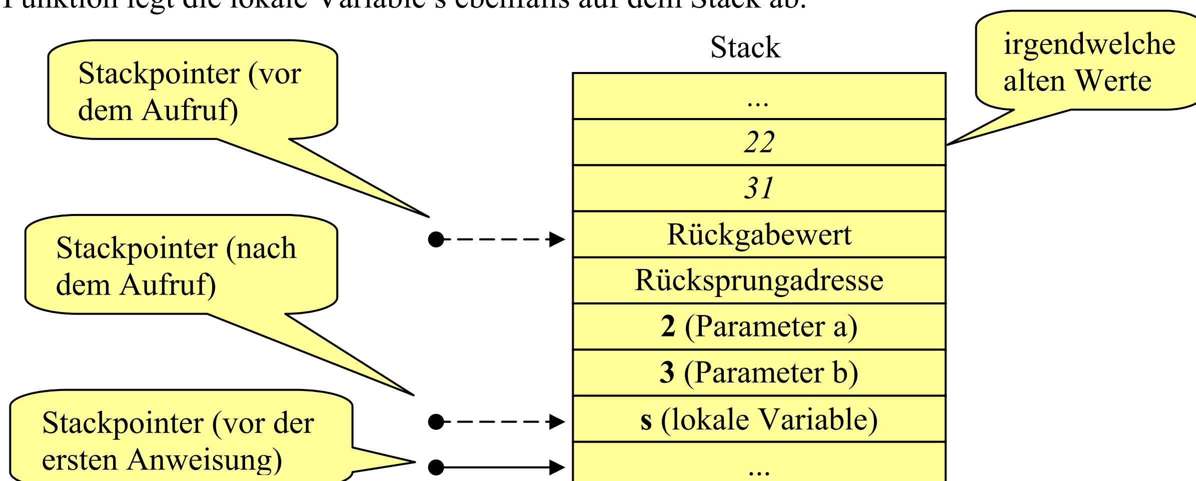
Wir wollen diese Funktion jetzt aufrufen:

```
int x = summe(1,summe(2,3));
```

zweimaliger Aufruf !

Was geschieht?

Zuerst muss der Wert des zweiten Parameters bestimmt werden, indem die Funktion *summe* mit den Werten *2* und *3* aufgerufen wird. Diese werden den formalen Parametern *a* und *b* zugeordnet. Zusätzlich werden (in einer festgelegten Reihenfolge) die Rücksprungadresse abgelegt und Platz für den Rückgabewert gelassen (weil eine Funktion aufgerufen wird). Die Funktion legt die lokale Variable s ebenfalls auf dem Stack ab.



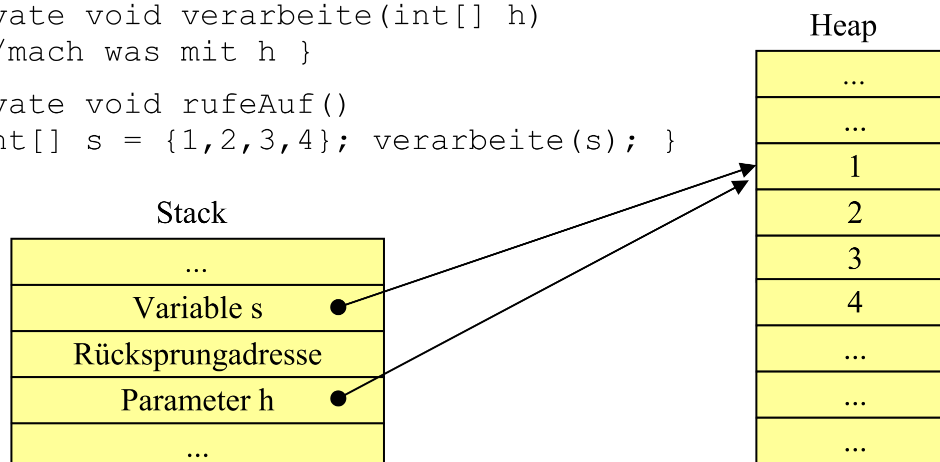
Die Funktion berechnet jetzt die Summe der Parameterwerte, weist das Ergebnis der Variablen *s* zu und kopiert diesen Wert auf den Platz des Rückgabewerts (bei der Ausführung von *return ...*). Der belegte Stackbereich wird wieder freigegeben durch Rücksetzen des Stackpointers. Das Programm wird an der Rücksprungadresse fortgesetzt. Da es sich um einen Funktionsaufruf handelte, zeigt der Stackpointer (in diesem Modell) jetzt auf den Rückgabewert.

Danach wird beim zweiten Aufruf dem Parameter *a* der Wert *1* und der Rückgabewert des ersten Aufrufs (*5*) (darauf zeigt der Stackpointer!) dem Parameter b zugewiesen, und es geht weiter wie oben. Es wird auch der gleiche Stackbereich neu belegt.

Werden Referenztypen als Parameter übergeben, dann sieht die Sache anders aus. Wir betrachten als Beispiel eine Methode mit einer Feldvariablen f, die eine zweite Methode mit dieser als Parameter aufruft:

```
private void verarbeite(int[] h)
{ //mach was mit h }

private void rufeAuf()
{ int[] s = {1,2,3,4}; verarbeite(s); }
```



Da die übergebenen Werte Referenzen sind, zeigen die beiden Größen *s* und *h* auf den gleichen Heapbereich. Änderungen an *h* wirken sich so auf *s* aus – ein typisches Verhalten für Referenzparameter. Will man das vermeiden, dann muss man ein „geklontes“ Objekt als Parameter übergeben. Da der Rückgabewert vom Typ *object* ist, muss man zusätzlich ein *Typecasting* machen:

```
private void verarbeite(int[] h)
{ //mach was mit h }

private void rufeAuf()
{
  int[] x, s = {1,2,3,4};
  x = (int[])s.clone();
  verarbeite(x);
}
```

Klonen mit Typecasting

5.3 Rekursionen

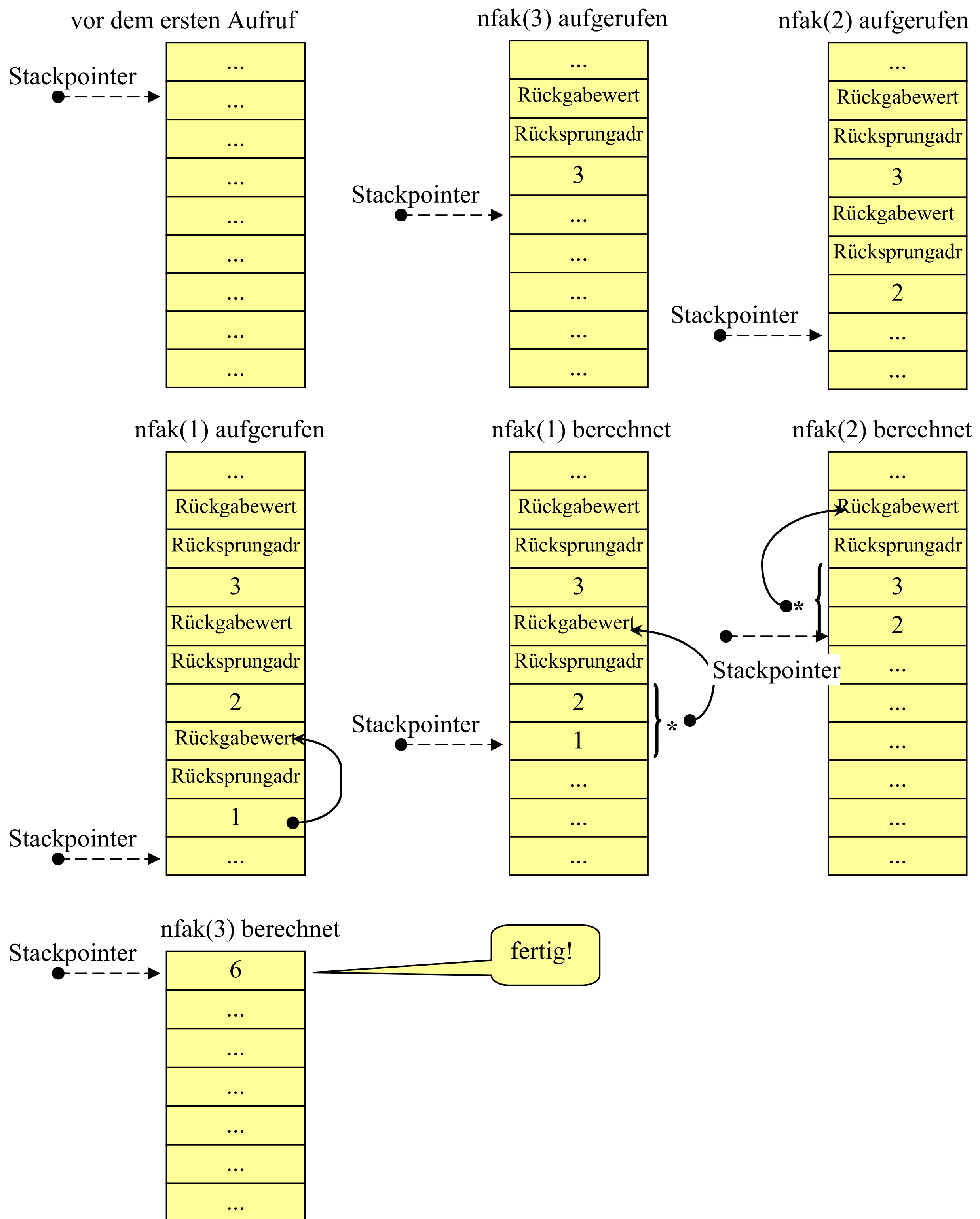
Die Arbeit mit einem Stapel macht aus Programmen außerordentlich mächtige Maschinen. Eine der Möglichkeiten ist die Fähigkeit von Methoden, sich selbst direkt oder indirekt aufzurufen. Stapelorientierte Sprachen können damit Rekursionen abarbeiten.

Das Standardbeispiel für eine rekursive Funktion ist die Berechnung der Fakultät einer natürlichen Zahl: $n! = n*(n-1)*(-2)*...*3*2*1$ Die Sonderfälle $0!=1!=1$ werden getrennt behandelt. Eine übliche rekursive Berechnungsvorschrift lautet nun: $n! = n*(n-1)!$ Die wollen wir implementieren:

```
private int nfak(int n){
  if(n==0) return 1;
  else if (n==1) return 1;
       else return n*nfak(n-1);
}
```

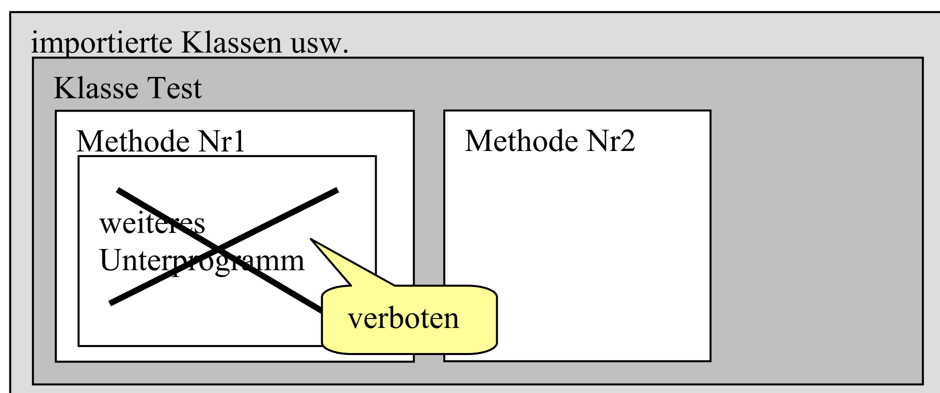
rekursiver Aufruf

Die Rekursion funktioniert, weil nach jedem Aufruf der Code mit einem neuen Satz von Werten auf dem Stack arbeitet. Wir sehen uns die Stackbelegungen für den Aufruf von `x=nfak(3);` einmal an:


vor dem ersten Aufruf
Stackpointer
...
nfak(3) aufgerufen
Rückgabewert
Rücksprungadr
3
nfak(2) aufgerufen
2
nfak(1) aufgerufen
1
nfak(1) berechnet
*
nfak(2) berechnet
nfak(3) berechnet
6
fertig!

5.4 Blöcke und Namensbereiche

Ein Block wird durch geschweifte Klammern eingeschlossen, die Anweisungen sowie Variablendeklarationen enthalten dürfen. Sind Namen im aktuellen Block vereinbart worden, zu dem ggf. auch die Parameterliste gehört, dann nennt man sie ***lokale Namen***. (Bei Variablen spricht man von lokalen Variablen.) Sind die Namen außerhalb des Block vereinbart, also in einer der äußeren Schachtelungen, dann sind sie für den Block ***global***. Die in einem bestimmten Programmabschnitt bekannten Namen, die an Konstanten, Klassen, Variable und Methoden vergeben wurden, bilden einen *Namensraum*. Innerhalb der Klassen bilden Methoden eigene Blöcke, die nebeneinander, also auf gleicher Ebene, deklariert werden. Eine weitere Verschachtelung von Methoden, wie in anderen Programmiersprachen möglich, ist in Java nicht erlaubt. Wird ein Name angegeben, dann sucht das System die dazu passende Größe „von innen nach außen". Der Compiler versucht zuerst, diesen unter den *lokalen Namen* des Blocks zu finden. Ist er dort nicht, dann sucht er unter den *globalen Namen* der Klasse oder Methode, und danach weiter außen bis zu den importierten Klassen.



Wird ein lokaler Name durch die gleiche Zeichenkette repräsentiert wie ein (zu diesem) globaler, dann *verdeckt* er den globalen Namen. Da dieses häufig mit Instanzvariablen von Klassen geschieht, kann man auf diese durch Voranstellung des Bezeichners *this* zugreifen.

```
public class test
{
  int i;
  private void m1()
  {
    int i;
    i = this.i;
    ...
  }
}
```

lokale Variable

Zugriff auf die Instanzvariable

6. Beispiel: Genetische Algorithmen

Der Einsatz genetischer Algorithmen ist in vielen Gebieten ein Standardverfahren zur stochastischen Lösung von Optimierungsproblemen. Nachgebildet wird ein evolutionäres Verfahren, in dem eine (meist) zufällige Anfangspopulation wiederholt den Operationen

- *Kreuzung* (aus dem existierenden „Genpool“) werden neue Individuen gebildet.
- *Mutation* (zufällig werden einige „Gene“ verändert)
- *Selektion* (mithilfe einer Bewertungsfunktion werden die geeignetsten Individuen ausgewählt)

ausgesetzt wird.

Wir wollen das Verfahren als Beispiel benutzen, in dem gehäuft Zeichenketten- und Feldoperationen angewendet werden. Als Anwendungsfall wählen wir das Problem, *Palindrome* zu erzeugen.

Palindrome sind Zeichenketten, die symmetrisch sind, also „von vorne“ oder „von hinten“ gelesen gleich lauten: ANNA ist ein Palindrom, PEPE nicht. Als Individuen wählen wir Zeichenketten mit 10 Zeichen, die wir in einem Feld namens *namen* unterbringen. (Der Bequemlichkeit halber nutzen wir die ersten beiden Strings zur Darstellung einer Überschrift, also nicht zum Speichern von Individuen.) Wir benutzen eine Konstante *nmax*, die die gewählte Größe der Population angibt. Ein zweites Feld *muell* kann neu erzeugte Individuen aufnehmen. Die Variable *generation* zählt die schon erzeugten Generationen.

eine Konstante (`final` ...)

```
final int nmax=25;

String[] namen = new String[nmax]; //die möglichen Palindrome
String[] muell = new String[nmax]; //Platz für neue Kombinationen
int generation = 0;                //zählt die Generationen
```

Zuerst müssen wir eine neue Anfangspopulation zufällig erzeugen. Dazu und an anderen Stellen brauchen wir ganzzahlige Zufallszahlen aus einem vorgegebenen Bereich. Wir führen eine Funktion zz ein, die genau dieses bewirkt:

die Parameter *von* und *bis* geben den Bereich der Zufallszahlen an.

```
private int zz(int von, int bis)
{
      int i=(int)(Math.abs(Math.round(Math.random()*(bis-von)+von)));
      return i;
}
```

Jetzt können wir einzelne zufällige Zeichenketten erzeugen, ebenfalls mithilfe einer Funktion:

```
private String zName()
{
      String neu = "";
      for (int i=0;i<10;i++) neu = neu + (char)zz(65,90);
      return neu;
}
```

zufällig gewählte Grossbuchstaben

Da wir keine doppelten Zeichenketten erzeugen wollen, müssen wir überprüfen, ob eine neu erzeugte Zeichenkette *s* im Feld *sf* schon vorhanden ist. Da auch diese Funktion an verschiedenen Stellen benutzt wird, geben wir den Bereich an, der überprüft werden soll.

```
private boolean schonDa(String s, String[] sf, int von, int bis)
{
        boolean gefunden = false;
        for (int i=von;i<=bis;i++) if(s.equals(sf[i])) gefunden = true;
        return gefunden;
}
```

Test auf inhaltliche Gleichheit

Mit diesen Hilfen können wir das Namen-Feld mit unterschiedlichen Zufalls-Zeichenkette füllen. Das Müll-Feld löschen wir gleich mit. ***Die Methode benutzt drei unterschiedliche Schleifenarten!***

```
private void neueNamen()
{
        int i;
        namen[2] = zName();
        i = 3;
        while(i<nmax)
        {
                do namen[i] = zName();
                while (schonDa(namen[i],namen,2,i-1));
                i++;
        }
        for (i=2;i<nmax;i++)
                muell[i] = new String("");
        generation = 0;
        anzeigen();
}
```

while-Schleife

do..while-Schleife

for-Schleife

Wie „mutiert“ man einen gegebenen String? Man ersetzt zufällig irgendwo einen Buchstaben durch einen neuen!

```
private String mutiere(String s)
{
        char c;
        int p;
        String neu;
        if (s.length()<10)
                s = zName();
        if(Math.random()>0.85)
        {
                p = zz(0,9);
                c = (char)zz(65,90);
                neu = s.substring(0,p)+c; fügen
                if (p<9) neu = neu+s.substring(p+1,10);
        }
        else neu = s;
return neu;
}
```

in Notfällen Zufalls-
namen neu erzeugen

in 15% der Fälle ...

... Platz im String erwürfeln ...

... und das Zeichen ebenso

Zeichen ersetzen

Damit können wir jetzt aus den bestehenden Zeichenketten neue erzeugen, indem wir sie z. B. verketten und einen 10-Zeichen-langen String irgendwo herausschneiden. Dieses Resultat wird dann in 15% der Fälle noch mutiert.

```
private void kreuzenMitMutation()
{
        int n1,n2,anf;
        String neu;
        for (int i=2;i<nmax;i++)
        {
                n1 = zz(2,24);
                n2 = zz(2,24);
                anf = zz(0,10);
                neu = namen[n1]+namen[n2];
                neu = neu.substring(anf,anf+10);
                neu = mutiere(neu);
                muell[i] = neu;

        }
}
```

das Müll-Feld mit Kombinationen füllen

Namen verketten, 10 Zeichen rausschneiden und ggf. mutieren

Es fehlt noch das Selektionsverfahren. Dazu bewerten wird alle alten und neu erzeugten Zeichenketten in Hinblick auf ihre „Palindromtauglichkeit“, d. h. wir ermitteln, wie viele Zeichen – von beiden Seiten gelesen – übereinstimmen.

```
private int bewerte(String s)
{
        int i=0,erg=0;
        boolean fertig=false;

        if (s.length()>9)
                do
                {
                        if (s.charAt(i)==s.charAt(9-i))
                        {
                                erg++;
                                i++;
                        }
                        else fertig=true;
                }
                        while (!fertig && (i<5));
        return erg;
}
```

Zeichen an beiden Seiten vergleichen

Jetzt können wir die alten Zeichenketten durch die geeignetsten aus allen vorhandenen ersetzen.

```
private void selektieren()
{
        String[] dieBesten = new String[nmax];
        String neu;
        int bew,bestwert=0,bestplatz=0;
        boolean besterInDenNamen=true;

        for (int i=2;i<nmax;i++)
        {
                do
                {
                        bestwert = -1;
                        bestplatz = 0;
```

die Besten!

das ganze Feld neu füllen

pessimistische Vorannahmen

alles durchsuchen

```
          for (int j=2;j<nmax;j++)
          {
                if (namen[j].length()>9)
                {
                      bew = bewerte(namen[j]);
                      if (bew > bestwert)
                      {
                            bestwert = bew;
                            bestplatz = j;
                            besterInDenNamen = true;
                      }
                }
```

ggf. die Position des neuen besten alten Werts merken

```
                if (muell[j].length()>9)
                {
                      bew = bewerte(muell[j]);
                      if (bew > bestwert)
                      {
                            bestwert = bew;
                            bestplatz = j;
                            besterInDenNamen = false;
                      }
                }
          }
```

ebenso bei den neuen

```
          if (besterInDenNamen)
          {
                neu = namen[bestplatz];
                namen[bestplatz] = "";
          }
```

Werte merken und löschen

```
          else
          {
                neu = muell[bestplatz];
                muell[bestplatz] = "";
          }
     }
     while((neu.length()<10) ||
          (schonDa(neu,dieBesten,2,i-1)));
```

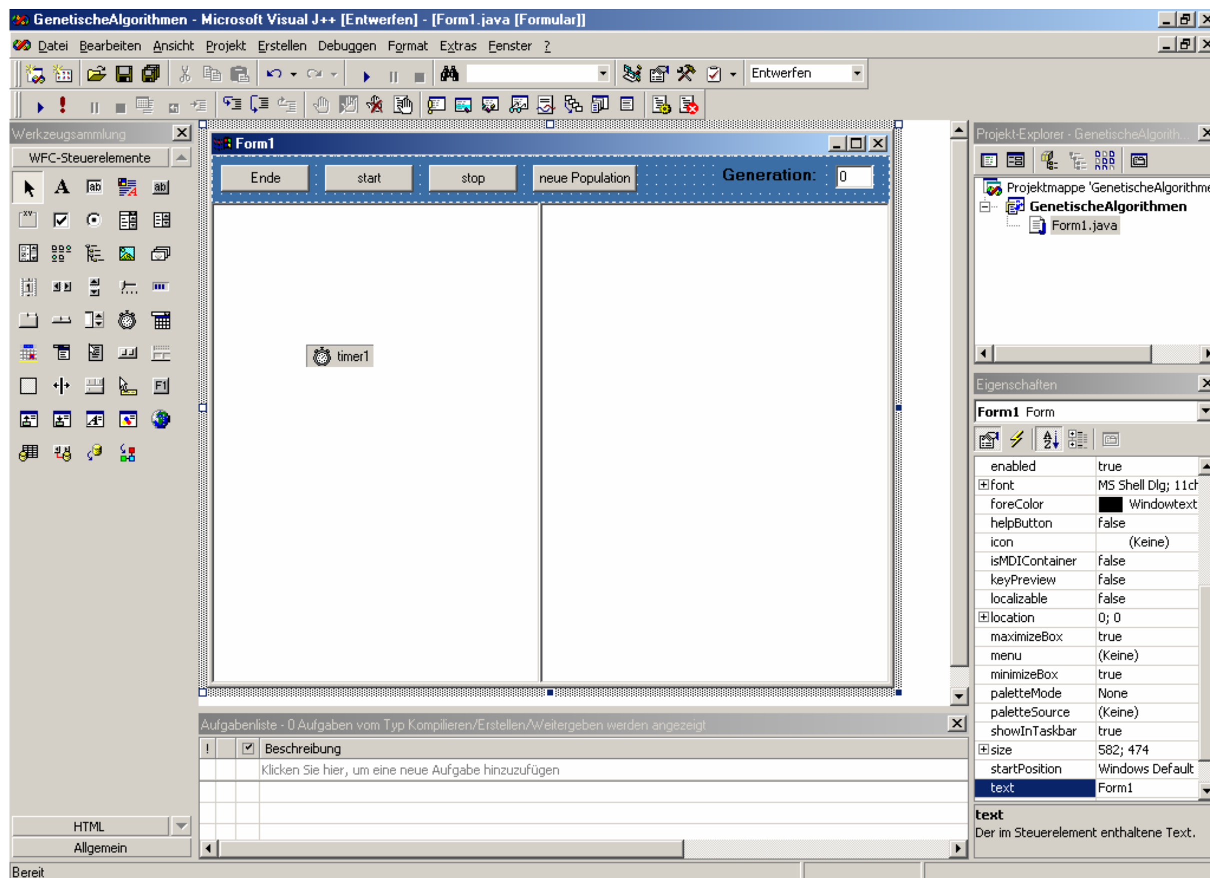
nur neue Werte suchen …

```
     dieBesten[i] = neu;
  }
namen = dieBesten;
}
```

… und einfügen

das sind die neuen Palindrom-Kandidaten

Für unser Programm benötigen wir eine geeignete Oberfläche. Die klicken wir uns unter J++ aus einer Panel-Komponente, einigen Buttons, einer Label-Komponente, einem Editierfeld zur Anzeige der Generation sowie zwei RichEdit-Komponenten zusammen, die die Zeichenketten anzeigen sollen.



Bemerkenswert ist der *Timer*: Da unser Programm einerseits „durchlaufen“ soll, um schnell neue Generationen zu berechnen, andererseits aber auch auf Mausklicks zu reagieren ist, fügen wir einen Timer ein, dessen Ereignisbehandlungsmethode periodisch in wählbaren Intervallen aufgerufen wird. In den „Pausen“ können dann andere Aktionen ausgelöst werden.

Unser Timer soll nun gerade die für genetische Algorithmen typischen Aktionen starten:

```
private void timer1_timer(Object source, Event e)
{
        kreuzenMitMutation();
        selektieren();
        generation++;
        anzeigen();
}
```

Der Timer kann von den Ereignisbehandlungsmethoden der Knöpfe gestartet bzw. gestoppt werden.

```
private void stop_click(Object source, Event e)
{
        timer1.stop();
}
```

```
private void start_click(Object source, Event e)
{
            timer1.start();
}
```

Für die Anzeige der Strings in den RichEdit-Komponenten bereiten wir die Zeichenketten etwas auf: Wir fügen in der Mitte ein Leerzeichen ein (dann lassen sich Palindrome besser erkennen) und schreiben die Anzahl der übereinstimmenden Zeichen bzw. „Treffer“ hinten ran.

```
private String[] bereite_auf(String[] s)
{
      String[] h = new String[nmax];
      int b;
      for (int i=0;i<nmax;i++)
      {
            h[i] = "";
            if(i<2) h[i]=s[i];
            else
                  if(s[i].length()>9)
                  {
                        h[i] = s[i].substring(0,5)+" "
                                          + s[i].substring(5,10);
                        b = bewerte(s[i]);
                        if (b==5) h[i] = h[i] + "   <--- Palindrom!";
                        else h[i] = h[i] + "   <--- (" + b + ")";
                  }
                else h[i] = s[i];
      }
      return h;
}
```

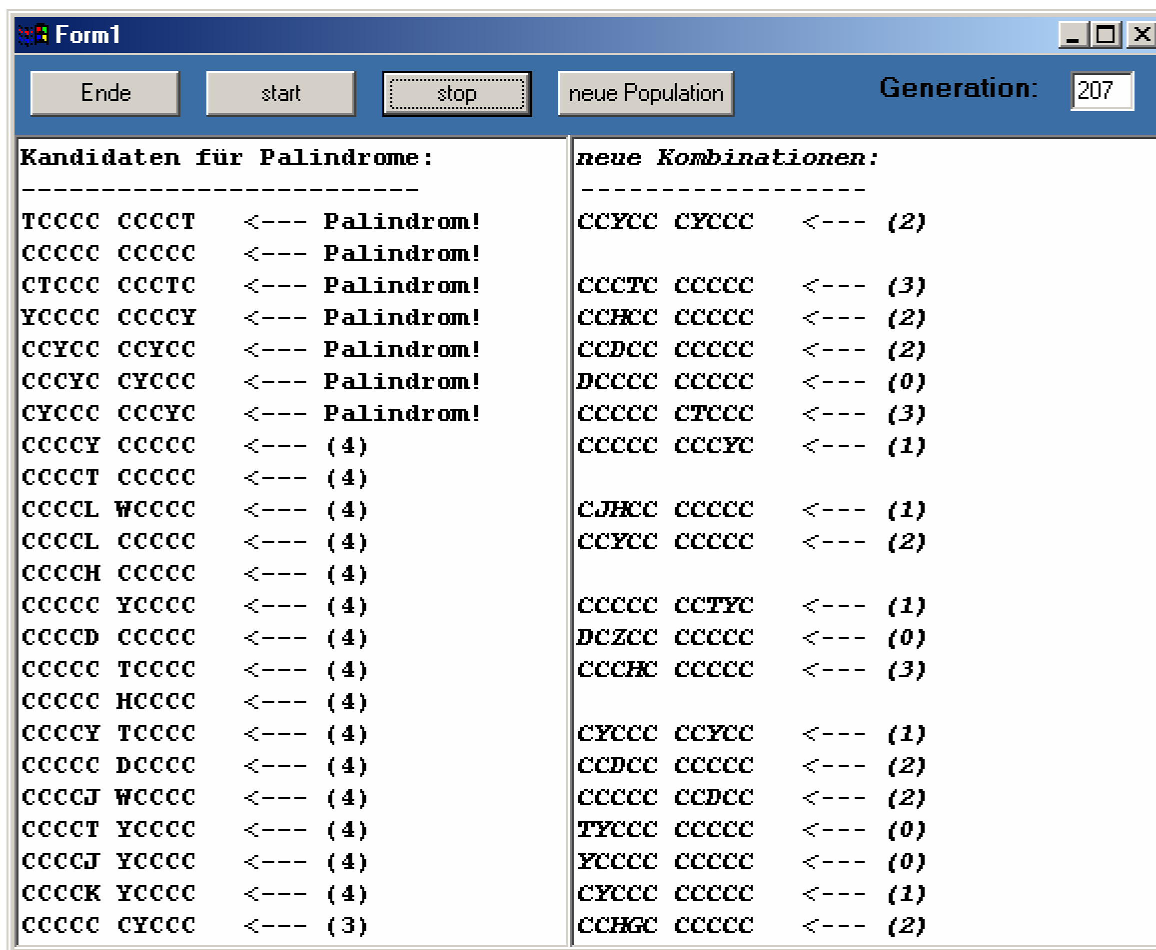
Leerzeichen einfügen

Kommentar anhängen

```
private void anzeigen()
{
      namen[0] ="Kandidaten für Palindrome:";
      namen[1] = "-------------------------";
      anzeige.setLines(bereite_auf(namen));
      muell[0] ="neue Kombinationen:";
      muell[1] = "------------------";
      hilfe.setLines(bereite_auf(muell));
      edit1.setText(""+generation);
}
```

Ergebnisse anzeigen

Das Ergebnis kann man auch besichtigen – hier nach 207 Generationen!



Aufgaben:

1. Das „Kreuzen" der Palindromkandidaten ist im Beispiel außerordentlich ungeschickt gemacht, da die „interessanten" Teile der zusammengesetzten Strings – also die Enden – meist abgeschnitten werden. Wählen Sie eine bessere Kreuzungsmethode!

2. Führen Sie eine Möglichkeit ein, während des Programmlaufs die „Umwelt" zu verändern. Beispielsweise könnten plötzlich keine Palindrome mehr, sondern Zeichenketten mit aufsteigend geordneten Zeichen gesucht sein.

3. Wenden Sie das Verfahren auf andere Situationen an:

   a: Gesucht sind die Koeffizienten ganzrationaler Funktionen vierter Ordnung, die „möglichst gut" durch einige vorgegebene Punkte laufen. Wieviel Punkte dürfen (müssen?) Sie eigentlich vorgeben? Was bedeutet hier „möglichst gut"?

   b: Lösen Sie Gleichungssysteme nach diesem Verfahren. Vergleichen Sie die Effizienz Ihrer Lösungen mit anderen Verfahren.