

VI. Programmieren im Großen – Strukturierter Entwurf und Unterprogramme

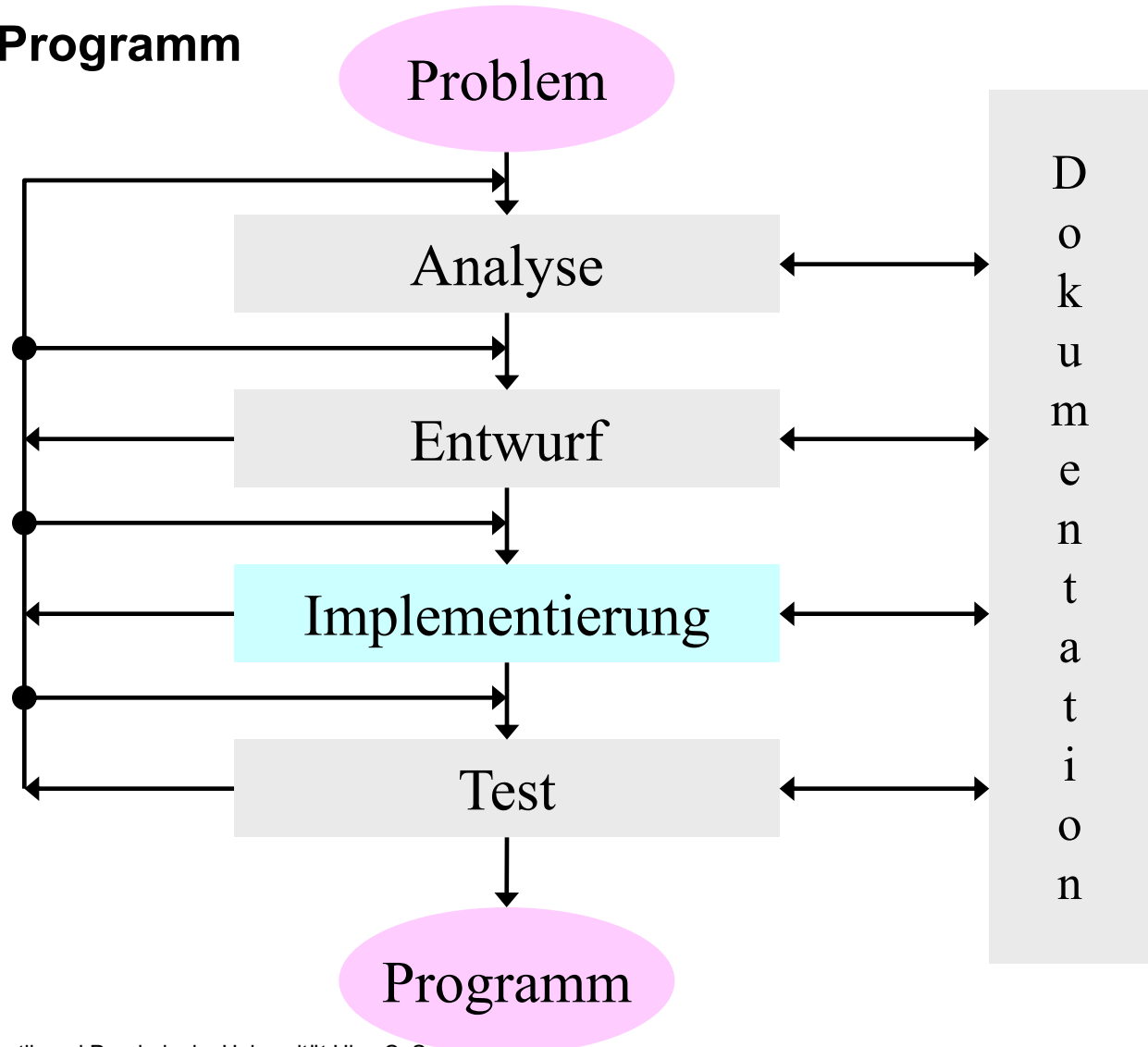
- 1. Aspekte der strukturierten Programmentwicklung und -wartung**
- 2. Methoden – Unterprogramme und Funktionen**
- 3. Parameterübergabe und Rückgabe von Ergebnissen**

1. Aspekte der strukturierten Programm-entwicklung und -wartung

- Phasen der Programmentwicklung
- Vom Problem zum Programm
- Wiederverwendung von Programmfragmenten
- Programmstrukturierung durch Unterprogramme

Phasen der Programmentwicklung

Vom Problem zum Programm



Phasen – Allgemein

Analyse

- Untersuchungen des Problems und des Problemumfelds
- Diskussion mit mehreren Personen (oft keine Informatiker, Arbeiten im Team)
- Fragestellungen / Tätigkeiten:
 - Problemstellung exakt und vollständig beschreiben
 - Gegebene Initialzustände und Eingabedaten bzw. –parameter bekannt?
 - gewünschte Endzustände, Ergebnisse und Ausgabewerte angeben
 - Randbedingungen, *Constraints*

Entwurf

- Entwicklung des Algorithmus
- kreativer Prozess (Auffassungsgabe, Kreativität, Erfahrung)
- Fragestellungen / Tätigkeiten:
 - existierende Lösungen für vergleichbare Probleme betrachten
 - allgemeinere Probleme suchen (*gibt es hierzu schon Lösungen?, Patterns*)
 - rekursive Aufteilung des Problems in Teilprobleme
 - Durchführung des Entwurfsprozesses für Teilprobleme
 - Zusammensetzen der Lösungen der Teilprobleme zur Lösung des Gesamtproblems

Implementierung

- Übertragung des Entwurfs in eine Programmiersprache
- Fragestellungen / Tätigkeiten:
 - Editieren
 - Übersetzung des Programms (Compilieren)

Test und Revision

- Überprüfung des Programms auf logische und technische Fehler - man kann i.A. nur die Existenz von Fehlern nachweisen, nicht die Abwesenheit!
- Fragestellungen / Tätigkeiten:
 - Korrektheit & Vollständigkeit
 - Fehlerbeseitigung, *Debugging*
- Teststrategien:
 - Testmengen konstruieren (Randfälle, Grenzwerte finden)
 - nach Fehlerbeseitigung erneut testen
- Wartung

Dokumentation

- Exakte Problemstellung und Beschreibung der generellen Lösungsidee
- Beschreibung des Algorithmus (evtl. in Pseudocode), Programmcode, Beschreibung der Testdatenmengen und Protokolle der Testläufe
- aufgetretene Probleme und alternative Lösungsansätze

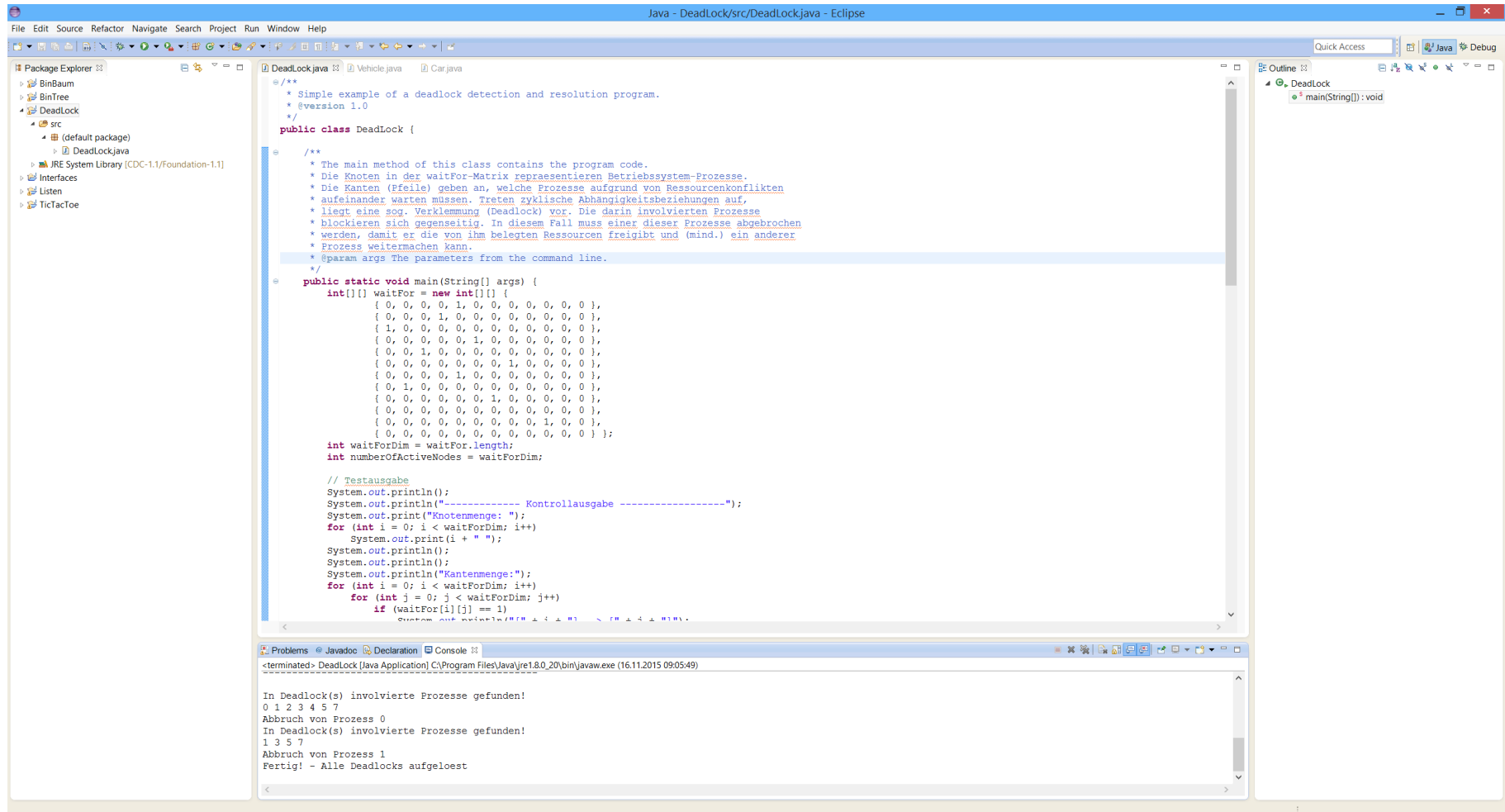
Werkzeuge für die Programmentwicklung

- **Editoren:**
 - Eingabe und Änderung (Manipulieren) des Programmcodes
 - häufig kontextsensitiv mit speziellen Eingabehilfen, z.B. Notepad++
- **Compiler:**
 - Transformation eines Quellprogramms in ein Zielprogramm
 - Syntaktische Überprüfung der Quelle
- **Interpreter:**
 - inkrementelle Abarbeitung des Quellcodes
 - Quellprogramm wird nicht übersetzt, sondern direkt in einer Umgebung ausgeführt
- **Debugger:**
 - Setzen spezieller Unterbrechungspunkte
 - Erkennung von Laufzeitfehlern
- **Dokumentationshilfen:**
 - Erstellung von Teilen der Dokumentation
- **Programmbibliotheken:**
 - Sammlungen fertig gestellter Programme
 - Bündelung von Programmen entsprechend bestimmter Funktionsgruppen (Ein-/Ausgabe, mathematische Funktionen, graphische Benutzerschnittstellen, etc.)

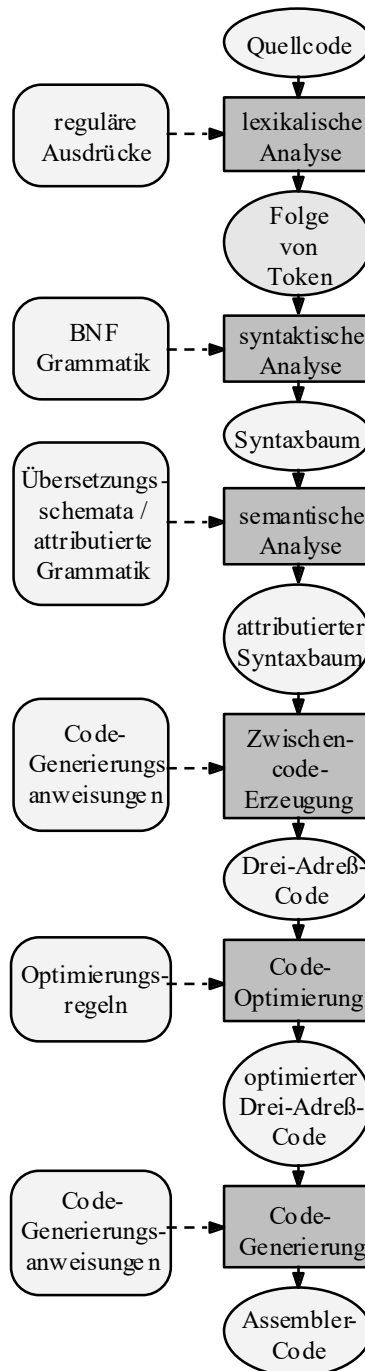
Oft zusammengefasst in *Integrated Development Environments* (IDEs) z.B. Eclipse

Integrated Development Environment (IDE)

Beispiele: **Eclipse** (<https://eclipse.org/>) oder **IntelliJ** (<https://www.jetbrains.com/de-de/idea/>)



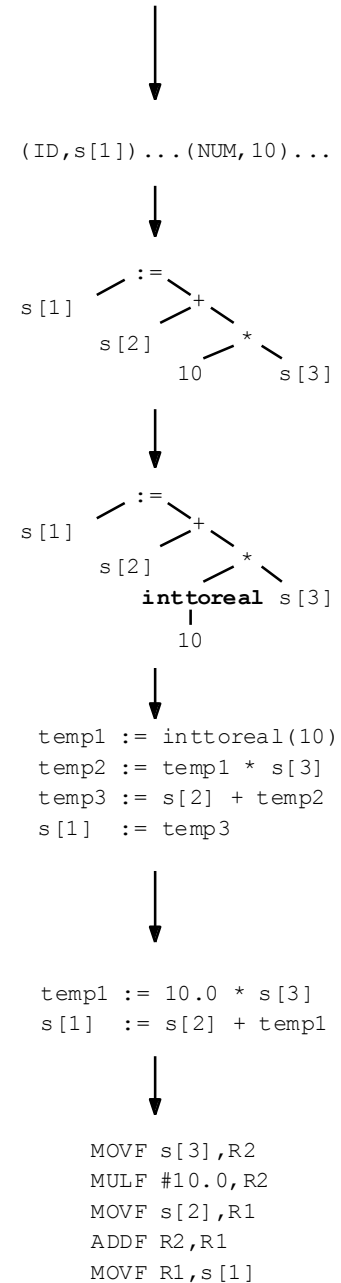
Programmentwicklung und -übersetzung



Symboltabelle s

1	a	real	
2	b	real	
3	c	real	
4			

a := b + 10 * c



Vom Problem zum Programm

Schrittweise Programmentwicklung

- **Vorbemerkung**

Es gibt keinen Algorithmus
für den Entwurf von Algorithmen

- Problemlösen ist ein **kreativer Prozess**, der ein umfassendes Verständnis der gestellten **Aufgabe**, Wissen über **Algorithmen** und **Techniken** sowie **Intuition** erfordert
- **Einordnung:**

„Wäre das Programmieren ein strikt deterministischer Prozess, der nach festen Regeln abläuft, so wäre es bereits seit langem automatisiert worden.“

(N. Wirth. Systematisches Programmieren. B.G.Teubner, Stuttgart, 1985)

Verschiedene Entwicklungsmethoden

Entwicklungsmethode

Top-down Strategie

- **Neukonzeption** von Algorithmen (mittels **schrittweiser Verfeinerung**)
- Ausgangspunkt ist meist eine individuelle Aufgabenbeschreibung

Bottom-up Strategie

- **Adaptation** eines bestehenden Programms an eine veränderte Zielsetzung
- Konstruktion eines bestehenden Programms **aus definierten Bausteinen** (Bibliothekslösungen)

Einordnung: Wir werden uns nachfolgend zunächst auf den Top-down Entwurf konzentrieren; **objektorientierte Methoden** folgen häufig einer Bottom-up Strategie

Pseudocode und schrittweise Verfeinerung anhand eines Beispiels

Das '3N + 1' Problem (Collatz-Problem; nach Lothar Collatz, 1937)

- Spezifikation des Problems

Gegeben sei eine positive ganze Zahl N . Berechne eine '3N+1'-Folge beginnend mit N wie folgt: Ist N eine gerade Zahl, dann teile N durch 2; wenn jedoch N ungerade ist, dann multipliziere N mit 3 und addiere 1 dazu. Fahre so lange fort, bis N gleich 1 wird.

Beispiel: Beginne mit $N = 3$; ist ungerade. Der Wert wird mit 3 multipliziert und 1 addiert, d.h. $N = 3 \cdot 3 + 1 = 10$. Da (der neue Wert) N gerade ist, wird die Zahl durch 2 geteilt: $N = 10/2 = 5$. Wir fahren entsprechend fort, der Prozess terminiert, wenn N die 1 erreicht; das ergibt die vollständige Folge: 3, 10, 5, 16, 8, 4, 2, 1.

- **Einordnung:** Das Problem ist für Mathematiker und Informatiker interessant, da es für die Frage bisher **keine Antwort gibt, ob die Berechnung der 3N+1 Folge** für beliebige Startwerte N nach **einer endlichen Anzahl von Schritten terminiert**.

Obwohl einige Folgen einfach zu berechnen sind, ist die Frage für den allgemeinen Fall bisher nicht beantwortet. Man kann also nicht sagen, ob die Berechnung ein **Algorithmus** ist, da die Forderung der Termination nach einer endlichen Anzahl von Verarbeitungsschritten nicht beantwortet werden kann.

(Anmerkung: In der Diskussion werden beliebige Werte für N angenommen, eine Variable vom Typ `int` in Java hat jedoch einen endlichen Wertevorrat!)

Allgemeiner Aufbau des Algorithmus zur Berechnung der Collatz-Funktion

■ Aufgabenbeschreibung

Schreibe ein Programm, das einen positiven Integer-Wert N (vom Benutzer) liest und daraufhin die '3N+1'-Zahlenfolge beginnend mit dem Integer-Wert ausgibt. Das Programm soll auch die Anzahl der Zahlen in der Folge zählen und diese ebenfalls ausgeben.

■ Erster Schritt: Die elementaren Verarbeitungsschritte (in Pseudocode)

```
Get a positive integer N from the user;  
Compute, print, and count each number in the sequence;  
Output the number of terms;
```

■ Pseudocode

- Verwendung **informeller Instruktionen**, die die Struktur einer Programmiersprache nachempfinden, jedoch
 - das genaue Detail und
 - die vollständige Syntax vernachlässigen
- verwende **Steueranweisungen** (Wiederholungen, Auswahl, Zuweisungen) wie in Programmiersprachen – jedoch informell –, um die Struktur zu spezifizieren

- Zweiter Schritt: (**schrittweise**) **Verfeinerung** des zentralen Abschnitts der Skizze
 - Es wird eine **Wiederholungsschleife** benötigt, da wir die Berechnung der Zahlen fortführen, bis die 1 erreicht wird
 - In einer **while**-Schleife wird die Berechnung wiederholt, solange die Zahl (der Folge) ungleich 1 ist

```

Get a positive integer N from the user;
while N is not 1:
    Let N := compute next term;
    Output N;
    Count this term;
endwhile
Output the number of terms;
  
```

Kommentare:

- Die **Einrückungen** sind Elemente der Pseudocode-Darstellung, die zur Strukturierung des Ablaufs dienen (sie entsprechen den Bereichen von Blöcken)
- Hier wird das Symbol ' := ' für die **Zuweisung** verwendet (häufig auch ' ← '); es unterscheidet sich von der in Programmiersprachen verwendeten Zuweisung '='!
- Die Wiederholung wird hier mit **endwhile** abgeschlossen, um ihr eine eindeutige Struktur zu geben; in Java ist dies nicht nötig, da die Syntax das Ende einer **while**-Anweisung eindeutig festlegt

- **Dritter Schritt: Berechne next term**
 - Es gibt eine **Fallunterscheidung**, was im nächsten Schritt berechnet werden soll – abhängig davon, ob N gerade oder ungerade ist
 - In einer **if**-Anweisung können die Berechnungszweige unterschieden werden

Get a positive integer N from the user;

while N is not 1:

if N is even:

 Let N := compute N / 2;

else

 Let N := compute 3 * N + 1;

endif

 Output N;

 Count this term;

endwhile

Output the number of terms;

Kommentar: Die **Fallunterscheidung** wird hier mit **endif** abgeschlossen, um die Struktur der Anweisungen eindeutig zu machen; in Java wird diese Endung nicht benötigt, da die Syntax das Ende einer **if**-Anweisung durch die Blockstrukturen eindeutig festlegt

- **Vierter Schritt: Zählen der Terme** (Count this term)
 - **Zählen** bedeutet hier in diesem Beispiel, dass man mit Null beginnt und bei jeder neuen Berechnung den Zähler um 1 erhöht
 - Es wird eine **Zähl-Variable** benötigt, um die Inkremente zu berechnen (dies ist ein allgemeines Programm-Muster, das in Programmen immer wieder benötigt wird)

```
Get a positive integer N from the user;
Let counter := 0;
while N is not 1:
  if N is even:
    Let N := compute N / 2;
  else
    Let N := compute 3 * N + 1;
  endif
  Output N;
  Let counter := counter + 1;
endwhile
Output the counter;
```

- Fünfter Schritt: Eingabe eines positiven Integer-Startwerts
 - Der Benutzer kann für N eine negative Zahl oder Null eingeben – *Was passiert dann?* (das Programm wird bei $N \leq 0$ endlos laufen, da nie $N = 1$ werden kann ...)
 - Man könnte dann den Prozess abbrechen, aber ein Programm sollte besser robust gegenüber fehlerhaften Eingaben sein! Lösung: Lies Eingaben so lange, bis der Benutzer eine positive Eingabe gibt

```

Ask user to input a positive number;
Let N be the user's response;
while N is not positive:
    Print an error message;
    Read another value for N;
endwhile
Let counter := 0;
while N is not 1:
    if N is even:
        Let N := compute N / 2;
    else
        Let N := compute 3 * N + 1;
    endif
    Output N;
    Let counter := counter + 1;
endwhile
Output the counter;

```

Kommentar: Die erste **while**-Schleife endet nur, wenn der Wert von N positiv ist. Ein **häufiger Programmierfehler** zu Beginn ist, eine **if**-Anweisung anstatt von **while** zu verwenden:

If N is not positive, ask the user to input another value

Die Lösung ist problematisch, wenn die zweite Benutzer-Eingabe ebenfalls negativ ist! Mit der **if**-Anweisung wird keine weitere Überprüfung der Eingabe mehr vorgenommen ...

Java-Programm mit der Implementierung des Algorithmus‘ (Demo: [ThreeN1.java](#))

```

public class ThreeN1 {
    public static void main(String[] args) {
        int N,          // eine positive Zahl (durch Benutzer eingegeben)
            counter;    // Zaehler, wieviele Zahlen berechnet werden

        /* -- Einlesen einer positiven Zahl ... */
        TextIO.put("Startpunkt fuer die Sequenz: ");
        N = TextIO.getlnInt();
        while (N <= 0) {
            TextIO.put("Der Startpunkt muss positiv sein. Bitte noch einmal ...: ");
            N = TextIO.getlnInt();
        }
        // an diesem Punkt ist N > 0 ...

        /* -- Berechnung der Folge von Zahlen und Ausgabe ... */
        counter = 0;
        while (N != 1) {
            if ((N % 2) == 0)    // N gerade?
                N = N / 2;
            else
                N = 3 * N + 1;
            TextIO.putln(N);
            counter = counter + 1;
        }

        /* -- Ausgabe der Anzahl der Zahlen in der Folge ... */
        TextIO.putln();
        TextIO.putln("Es waren " + counter + " Terme in der Folge");
    } // end main
} // end class ThreeN1

```

Anmerkungen zum Algorithmus ...

Zum Bereich der Zählvariable

- Der erste Wert der Folge – der durch den Benutzer eingegebene Anfangswert der Zählvariable N – wird nicht ausgegeben und nicht gezählt
- *Ist das ein Fehler? War die Spezifikation genau genug, um hierzu eine Antwort zu geben?*

Eine **Lösung** kann hier einfach gefunden werden:

Ersetze im Lösungsprogramm

```
:  
  
counter = 0;  
while (N != 1) {  
    :  
}
```

durch

```
:  
  
TextIO.putln(N); // drucke den Initial-Wert von N  
counter = 1;     // zaehle diesen Wert mit zur Folge  
while (N != 1) {  
    :  
}
```

Bemerkungen zum *Top-Down* Entwurf und schrittweiser Verfeinerung

- Es existiert häufig nicht die beste oder alleinige Lösung für ein algorithmisches Problem – die Lösung hängt von bestimmten Randbedingungen ab, auch der Entscheidung des Entwicklers / Programmierers
- Ein Entwickler muss **Erfahrungen** besitzen, um **Entscheidungen zu fällen** und diese auch zu **hinterfragen ...**

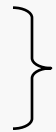
„Students must be taught to be conscious of the involved decisions and to critically examine and reject solutions, sometimes even if they are correct as far as the result is concerned; they must learn to weigh the various aspects of design alternatives in the light of these criteria. In particular, ... to revoke earlier decisions, and to go back, if necessary, to the top.“

(N. Wirth. Program development by stepwise refinement. Communications of the ACM **14**(4), 221-227, 1971)

- Es gibt **Programmiersprachen**, in denen die **schrittweise Verfeinerung explizit** in der Sprache als Anweisungsformat unterstützt wird!

Bsp.: ELAN (***E**ducational **L**anguage*; eine Schulsprache, die an der TU Berlin entwickelt wurde) – die Verfeinerungen funktionieren wie Makros, die automatisch als Codefragmente ersetzt werden

```
init stapel;
werte auf stapel legen;
werte vom stapel holen.
```



primärer Programmabschnitt

```
init stapel:
...
```



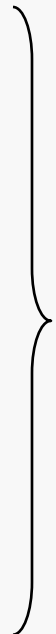
```
werte auf stapel legen:
INT VAR anzahl :: 0,
      wert;
```



```
REP
  get(wert);
  push(wert);
  anzahl INCR 1
  UNTIL ende kriterium
END REP.
```



```
werte vom stapel holen:
...
```



Verfeinerungen

Wiederverwendung von Programmfragmenten

Redundanter Programmcode

Einordnung

- Oftmals muss in einem Programm an verschiedenen Stellen eine ähnliche Aufgabe gelöst bzw. eine Problemlösung berechnet werden

Bsp.: Wir benötigen an mehreren Stellen eine Berechnung des ggT, eine Primfaktorenzerlegung, die Sortierung von Daten, eine Zyklussuche, ...

- **Lösungsansätze und -alternativen** (basierend auf dem jetzigen Kenntnisstand)

Einfügen des entsprechenden Programmcodes an allen betreffenden Stellen, an denen das Codesegment auftritt ...

- mittels explizitem wiederholtem „Ausprogrammieren“ der Lösung
- mittels *Copy & Paste* von Programmfragmenten – die separat getestet wurden – und deren lokale Anpassung

Variante 1 – Explizites Ausprogrammieren

Bsp.: Die ggT-Berechnung wird an verschiedenen Stellen im Programm benötigt

```

:
int c, d;
:
// berechne ggT(c, d);

```

```

while (c != d) {
    if (c > d)
        c = c - d;
    else
        d = d - c;
}

```

```

:
int e, f;
:
// berechne ggT(e, f);

```

```

while (e != f) {
    if (e > f)
        e = e - f;
    else
        f = f - e;
}

```

```

:
int g, h;
:
// berechne ggT(g, h);

```

```

while (g != h) {
    if (g > h)
        g = g - h;
    else
        h = h - g;
}

```

Ansatz/Vorgehen:

- Die Abschnitte werden an den Stellen der Kommentare direkt programmiert
- Es werden die jeweiligen Variablen verwendet

Probleme:

- Programmcode wird aufgebläht und damit unübersichtlich
- erhöhtes Risiko von Programmierfehlern
- Erhöhter Wartungsaufwand

Bsp.: Algorithmus soll durch einen effizienteren ersetzt werden ... das muss per Hand an allen Stellen erfolgen

Variante 2 – Copy & Paste mit Anpassungen

```

:
int c, d;
:
// berechne ggT(c, d);

```

```

while (c != d) {
  if (c > d)
    c = c - d;
  else
    d = d - c;
}

```

```

:
int e, f;
:
// berechne ggT(e, f);

```

```

while (e != f) {
  if (e > f)
    e = e - f;
  else
    f = f - e;
}

```

```

:
int g, h;
:
// berechne ggT(g, h);

```

```

while (g != h) {
  if (g > h)
    g = g - h;
  else
    h = h - g;
}

```

Ansatz/Vorgehen:

- Der Programmcode wurde separat implementiert und getestet
- wird jetzt hier hineinkopiert
- die Variablennamen werden angepasst

```

while (va != vb) {
  if (va > vb)
    va = va - vb;
  else
    vb = vb - va;
}

```

Einordnung:

- Etwas robustere Variante als die explizite Programmierung
- Keine wesentliche Verbesserung, da Aufwand für Fehlerbehandlung und Wartung weiterhin hoch ist

Vermeidung von redundantem Programmcode

- **Identifizierte Probleme** mit den bisherigen Ansätzen:
 - Umfang und Unübersichtlichkeit von Lösungen
 - Programmierfehler
 - Aufwand bei Erstellung und Wartung
- **Bessere Alternative**
 - Verwendung **separater Codeelemente**, die durch **Aufruf** verwendet werden können
 - Diese werden einmal entwickelt, getestet und dann immer wieder verwendet (*Code re-use*)
- **Neue Fragen**, die hierbei entstehen ...
 - *Wie gelangt man in das Re-Use Programmfragment?*
 - Die Variablen an der „Auftraggeberstelle“ heißen (oft) anders als im Re-Use Programmfragment – *Wie kann man die gewünschten Variablen dorthin bringen?*
 - *Wie weiß das Re-Use Programmfragment, wohin nach Abarbeitung der Aufgabe zurückgekehrt werden muss?*

Variante 3 – Wiederverwendbare Programmabschnitte

```

:
int c, d;
:
// berechne ggT(c, d);
<Springe zu Fragment>

```

```

:
int e, f;
:
// berechne ggT(e, f);
<Springe zu Fragment>

```

```

:
int g, h;
:
// berechne ggT(g, h);
<Springe zu Fragment>

```

```

while (va != vb) {
  if (va > vb)
    va = va - vb;
  else
    vb = vb - va;
}

```

Ansatz/Vorgehen:

- Der separate Programmabschnitt wird ausgeführt
- Die Variablen (c, d), (e, f), (g, h) müssen jeweils dort verwendet werden

Zu lösende Fragen:

- *Wie werden die Variablen im Re-Use Programmteil eingesetzt?*
- *Wie wird der Programmabschnitt angesprungen?*
- *Wie wird wieder zurück gesprungen?*

Programmstrukturierung durch Unterprogramme

Wiederverwendung von Programmteilen

Verwendung separater Programmabschnitte mit Sprüngen

- Die Detailanalyse zur Ausführung des separaten (*Re-Use*) Programmabschnitts muss schrittweise die **neu aufgeworfenen Fragen** beantworten:
 - Lösung des Problems der **Übergabe von Variablen** an den Programmabschnitt
 - Lösung des Problems des **Anspringens** des Programmabschnitts
 - Lösung des Problems des **Rücksprungs** aus dem Programmanschnitt
- Kommentar:

Die einzelnen Fragestellungen werden nacheinander betrachtet und hierfür konzeptionelle Lösungen skizziert – diese berühren **strukturelle Aspekte der Programmkonstruktion** und des **Algorithmenentwurfs** allgemein

Variante 3 – Wiederverwendbare Programmabschnitte (recap)

```

:
int va, vb;
int c, d;
:
// berechne ggT(c, d);
va = c;
vb = d;
<Springe zu Fragment>
// nach Rueckkehr
c = va;
d = vb;

```

```

:
int e, f;
:
// berechne ggT(e, f);
va = e;
vb = f;
<Springe zu Fragment>
// nach Rueckkehr
e = va;
f = vb;

```

```

while (va != vb) {
    if (va > vb)
        va = va - vb;
    else
        vb = vb - va;
}

```

Lösung des Variablenproblems:

- Vor dem Sprung (Vorbereitungsabschnitt):
lokale Variablen $\xrightarrow{\text{copy}}$ Variablen in *Re-Use*
- Nach Rückkehr aus *Re-Use* Fragment:
Variablen in *Re-Use* $\xrightarrow{\text{copy}}$ lokale Variablen

Weiterhin zu lösende Fragen:

Es bleibt noch das Problem

- des **Ansprungs in das Fragment** und
- des **Rücksprungs** an die Auftragsstelle zu lösen

Bedingte Sprünge und Marken

```

:
int va, vb;
int rucksprung;
int c, d;
:
// berechne ggT(c, d);
va = c;
vb = d;
rucksprung = 1;
goto ggt;
// nach Rueckkehr
M1: c = va;
d = vb;

```

```

:
int e, f;
:
// berechne ggT(e, f);

```

Hinweis: In Java so nicht realisierbar!

ggt:

```

:
while (va != vb) {
    if (va > vb)
        va = va - vb;
    else
        vb = vb - va;
}

```

```

if (rucksprung == 1)
    goto M1;
else if (rucksprung == 2)
    goto M2;
:

```

Lösung des Sprung-Problems:

- „Hinsprung“
 - Verwendung von Marken (*Labels*) und
 - Sprunganweisung (*goto*)
- „Rücksprung“
 - Je nach Programmiersprache gibt es verschiedene Möglichkeiten
 - Eine Lösung, die immer funktioniert, wenn *goto* unterstützt wird
 - Marken an den Rücksprungstellen
 - „Merker“, welche Marke für den Rücksprung gewählt werden soll
 - Bedingtes *goto* für den Rücksprung

Insbesondere wenn viele Variablen vor dem „Sprung“ mit Werten versorgt werden müssen oder das Resultat aus mehreren Variablen übernommen werden muss

Kurze Bewertung und Weiterführung

Fazit: Insgesamt ist diese **Lösung recht kompliziert** und damit fehleranfällig

- **Was verursacht die Komplexität dieser Lösung?**
 - Diverse Zuweisungen an Variablen sind erforderlich, um vom „Aufruf“-Kontext auf den *Re-Use* Kontext abzubilden und umgekehrt
 - Explizites „Merken“ von Rücksprungzielen ist erforderlich
 - Viel explizites Hin- und Herspringen im Programm (mit *goto*)
- Lösungsidee:
 - Entkopplung der Namen im *Re-Use* Teil von den Variablen-Namen im „Aufruf“-Kontext (Realisierung des *Re-Use* Teils als *black box*)
 - Das Programm sorgt selbst für die Hinterlegung der jeweiligen Rücksprungstelle
 - ... und springt nach Beendigung des *Re-Use* Teils an diese zurück
- Realisierung der **Lösung: Unterprogramme**
 - Prozeduren
 - Funktionen
 - (Methoden)

Hinweis: In der **maschinennahen Programmierung** (**Technische Informatik**) werden die Konzepte dieser *Re-Use* Variante noch einmal diskutiert!

Unterprogramme

Allgemeine Bemerkungen und Einordnungen

- Die Definition von **Unterprogrammen** stellt einen **Abstraktionsschritt** bei der Programmentwicklung dar
- Mit **Unterprogrammen** werden **Methoden zur Berechnung** bzw. **Lösung von Teilaufgaben** formuliert
- Struktur der Definition eines Unterprogramms

Name : _____

Eingabe (*input*) : _____ (**Aufruf**-Parameter des Unterprogramms)

Ausgabe (*output*) : _____ (Ergebnis der Berechnungen durch
Rückgabe-Parameter oder einen
Rückgabewert)

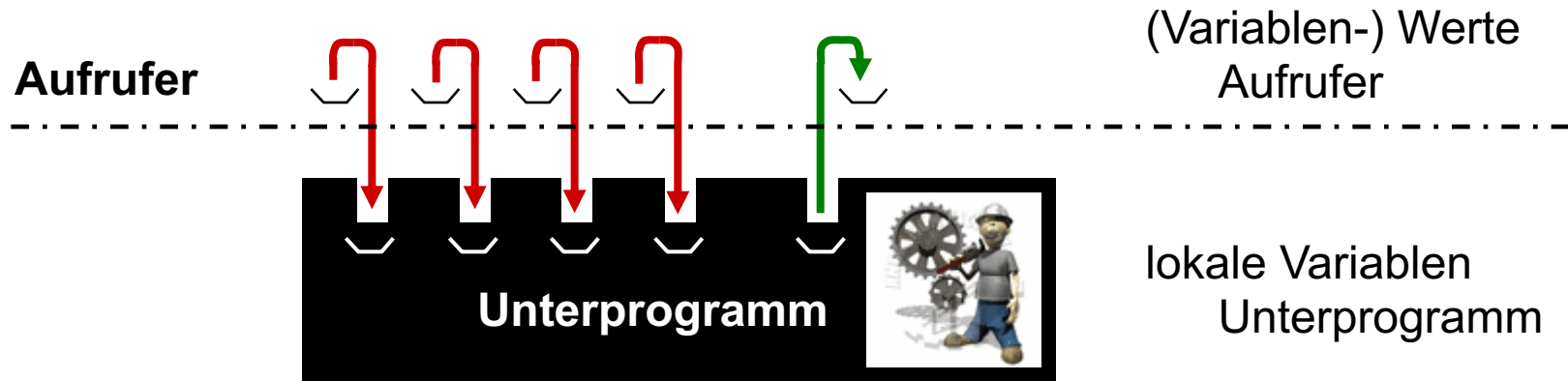
...

...

...

Rumpf des Unterprogramms

Grundkonzept von Unterprogrammen (idealisiert)



- Der **Aufruf** eines **Unterprogramms** erfolgt unter Angabe dessen **Namens** im aufrufenden Programmabschnitt; der **Aufrufer** braucht die interne **Realisierung des Unterprogramms** nicht zu kennen (*black box* und *information hiding* Prinzip)
- Wenn das **Unterprogramm** Eingabedaten erwartet, muss der **Aufrufer** die für die Aufgabe **relevanten Daten** an das **Unterprogramm** übergeben
- Das **Unterprogramm** kann mit den übergebenen Daten die Aufgabe selbständig und unabhängig bearbeiten
- Nach der Bearbeitung der Aufgabe gibt das **Unterprogramm** (**falls vorgesehen**) die **Ergebnisse** an den **Aufrufer** zurück

Unterprogramme – Arten, Aufruf, Parametrisierung, Ergebnisse

▪ Unterscheidung der **Art eines Unterprogramms** bzgl. **Eingabe** und **Ausgabe**

- **Eingabe:** Unterprogramm ...
 - **ohne** Eingabe
 - **mit** Eingabe (formale Parameter)
- **Ausgabe:** Unterprogramm ...
 - **ohne** Ausgabe
 - **mit** Ausgabe über **Parameter** (Rückgabe-Parameter) ¶
 - **mit** Ausgabe über **Funktionswert** (ähnlich der Mathematik)

▪ Bsp.: (in **Pseudocode** – keine echte Java-Syntax)

¶ in **Java nicht** vorgesehen

```
procedure printHallo() {
    println("Hallo"); }
```

```
procedure printMsg(in String msg) {
    println(msg); }
```

```
procedure compSum(in int a, in int b, out int sum) {
    sum = a + b; }
```

```
int function compSumF(in int a, in int b) {
    compSumF = a + b; }
```

Viele PS unterscheiden bei der Parameterübergabe zwischen

- **call by value** (hier: **in**)
= **mono-direktional**
(Wert geht nur in das UP hinein)
- **call by reference** (hier: **out**)
= **bidirektional**
(Wert kann via Parameter (auch) zurückgeliefert werden)

In **Java**: nur „call by value“!

Realisierung in Java

Pseudocode-Notation	<u>Java</u>
<code>procedure printHallo() { println("Hallo"); }</code>	<code>public static void printHallo() { System.out.println("Hallo"); }</code>
<code>procedure printMsg(in String msg) { println(msg); }</code>	<code>public static void printMsg(String msg) { System.out.println(msg); }</code>
<code>procedure compSum(in int a, in int b, out int sum) { sum = a + b; }</code>	** geht in <u>Java</u> nicht ! ** (<u>aber</u> Übergabe von Zeigervariablen)
<code>int function compSumF(in int a, in int b){ compSumF = a + b; }</code>	<code>public static int compSumF(int a, int b) { return a + b; }</code>

Erläuterungen:

- **call-by-value** besagt für einen **Parameter** lediglich, dass etwaige Änderungen am Inhalt (Wert) dieses Parameters im **Unterprogramm** nicht an den **Aufrufer** zurück gereicht werden
- Wenn der **Parameter** eine **Zeigervariable** (also eine Referenz auf ein Objekt) ist, dann werden Änderungen dieses Parameters selbst (= Zeigervariable) im **Unterprogramm** (auch auf `null` setzen!) ebenfalls nicht an den **Aufrufer** zurück gereicht
- **ABER:** der Inhalt des durch einen **Zeiger referenzierten Objekts** kann jedoch verändert werden!

Demo: UnterprogrammTest.java

Beispiel zur Parameterübergabe (in Java):

```

class UnterprogrammTest {
    public static void main(String[] args) {
        int a = 1; // einfache Integer-Variable
        int[] b = new int[] { 1 }; // Zeiger-Variable (auf int-Array)
        System.out.println("Vorher: a = " + a + " / b[0] = " + b[0]);
        change1(a);
        change2(b);
        System.out.println("Nachher1: a = " + a + " / b[0] = " + b[0]);
        change3(b);
        System.out.println("Nachher2: b[0] = " + b[0]);
    }
    public static void change1(int x) {
        x = 200;
    }
    public static void change2(int[] x) {
        x[0] = 200;
    }
    public static void change3(int[] x) {
        x = null;
    }
}

```

```

<terminated> UnterprogrammTest [Java Application] (
→ Vorher: a = 1 / b[0] = 1
→ Nachher1: a = 1 / b[0] = 200
→ Nachher2: b[0] = 200

```

Fazit, Einordnung und Ausblick

- **Schrittweise Verfeinerungen** werden in gängigen Programmiersprachen nicht direkt durch sprachliche Konstrukte unterstützt
- Jedoch dienen **Unterprogramme** – **auch wenn sie nur an einer Stelle aufgerufen werden** – der **Abstraktion** und **Strukturierung** eines Programms

„You should not hesitate, however, from formulating an action as a procedure – even when called only once – if done so enhances the readability of a program. ... Defining development steps as procedures makes a more communicable and verifiable program.“

(K. Jensen, N. Wirth. Pascal User Manual and Report, 3rd edition. Springer, New York, 1985, p.106)

- **Unterprogramme** – **Funktionen** oder **Methoden** (in Java) – stellen elementare Mechanismen zur funktionellen Abstraktion bei der Objektorientierung zur Verfügung

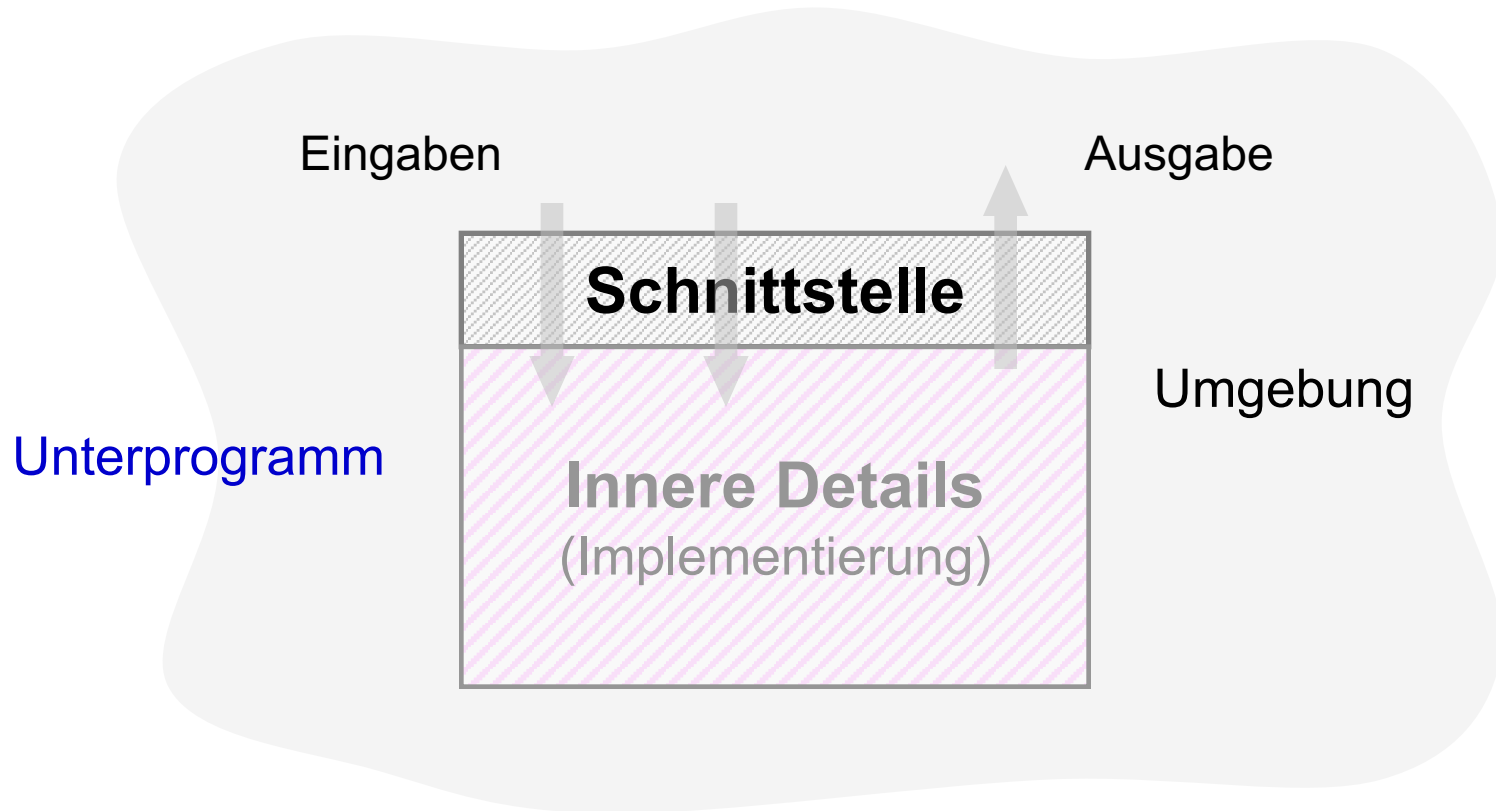
2. Methoden – Unterprogramme und Funktionen

- Motivation
- Statische Unterprogramme und Funktionen
- Lokale Variable und Klassen-Variable
- Statische Methoden und Variablen

Motivation

Unterprogramme als *Black boxes*

- **Unterprogramme** sind Instruktionsfolgen, die eine bestimmte Aufgabe lösen und unter einem bestimmten **Namen** identifiziert werden
- *Black box* (vgl. schrittweise Verfeinerung):
 - Die Inhalte werden **vor äußerem Einblick verborgen** (man kann in die Kiste nicht hinein sehen; hier: die Details der Implementierung sind nicht sichtbar)
 - Die Interaktion des Unterprogramms mit der Umgebung (Außenwelt) erfolgt über eine **Schnittstelle** (*interface*) – Angabe der Eingaben und Ausgaben
- Allgemeine Regeln für das *Design* eines Unterprogramms i.S. einer *Black box*:
 - I. **Schnittstelle** sollte möglichst einfach, klar strukturiert und verständlich sein
 - II. Alles was zur **Verwendung** der *Black box* notwendig ist, sollte **über die Schnittstelle** bekannt gemacht werden – keine Implementierungsdetails
 - III. Der Entwickler sollte nichts über das größere System wissen müssen, in dem die *Black box* verwendet wird



- **Spezifikation einer Schnittstelle:** Festlegung was die *Black box* leistet und wie sie durch die Elemente der Schnittstelle (**Parameter**) gesteuert werden kann
- **Vertrag (contract) des Unterprogramms:** Semantische und syntaktische Definition der Schnittstelle

Typen von Unterprogrammen

Unterprogramme (Methoden)

```
graph TD; A[Unterprogramme (Methoden)] --> B[ohne Rückgabe]; A --> C[mit Rückgabe];
```

ohne Rückgabe
(Prozeduren ohne Ergebniswert)

mit Rückgabe
(Funktionen)

Statische Unterprogramme und Funktionen

Definition von Unterprogrammen in Java

Vorbemerkungen

- Jedes **Unterprogramm** in Java muss **innerhalb einer Klasse** definiert werden

Hinweis: Dies ist **anders als in vielen anderen Programmiersprachen**, in denen Unterprogramme (Prozeduren, Funktionen) frei und unabhängig definiert werden können! (Unterprogramme werden so in Java **thematisch gebündelt**)

Zielsetzung in Java:

- Gruppierung von zusammengehörigen Unterprogrammen und Variablen in Klassen zur besseren Strukturierung
- Java ist eine vom Entwurf **objektorientierte Sprache**, in der **Unterprogramme Methoden** bilden, die das **Verhalten** definieren – zusammen mit (Klassen-) **Variablen** beschreiben sie den **Zustand** von Objekten
- **Klassenvariablen („statische Variablen“)** werden **innerhalb einer Klasse separat** (d. h. außerhalb von Unterprogrammen) **deklariert**

Struktur

Allgemeine Form einer **Unterprogramm-Definition** in Java

```
<Modifizierer>
  <Rueckgabe-Datentyp> <Unterprogramm-Name> (<Parameter-Liste>) {

    <Anweisungen>          Rumpf des Unterprogramms

  }
```

Klammern { } zur Festlegung des **Blocks**

Erläuterungen:

- Ein **Beispiel** für diese **Struktur** haben wir bereits kennen gelernt:

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

Hinweis: Das Beispiel entspricht vom Aufbau her einem Unterprogramm:

- Es gibt die **<Modifizierer>** **public** und **static**,
- Der **<Rueckgabe-Datentyp>** ist **void** (leer), d.h. es wird **kein Ergebnis** geliefert

- Der **<(Unter-) Programm-Name>** ist `main`,
 - Die **<Parameter-Liste>** besteht hier aus einem Argument vom *Array*-Typ `String[]` mit dem Bezeichner (Name) `args`
- **<Modifizierer>** können zu Beginn der Definition auftreten (jedoch auch leer bleiben); sie legen bestimmte Eigenschaften des Unterprogramms fest
 - Der **<Rückgabe-Datentyp>** legt den Datentyp fest, den das Unterprogramm als Ergebnis liefert
 - wird **kein Ergebniswert** geliefert, so wird dies durch das Schlüsselwort `void` angezeigt (in der imperativen Programmierung werden diese Unterprogramme **Prozeduren** genannt)
 - wird ein **Ergebniswert** geliefert, so gibt das Schlüsselwort den Datentyp des Resultats an (Unterprogramme mit Resultat werden **Funktionen** genannt)

Bemerkungen: Man kann sagen, dass mit `void` ein leerer Rückgabedatentyp geliefert wird und damit alle Unterprogramme letztlich **Funktionen** sind – wir werden diese Begriffe daher synonym verwenden!

Im Kontext der **Objektorientierung** werden Funktionen als **Methoden** bezeichnet mit denen der **Zustand eines Objekts** verändert bzw. ausgelesen werden kann! Wir werden in der Folge schon jetzt vereinheitlichend von **Methoden** sprechen!

Methoden – Parameter und Rumpf

- **Methoden** (Funktionen) werden (syntaktisch) stets durch “()” identifiziert
- Die **<Parameter-Liste>** in () kann ...
 - **leer** sein (die Methode erhält keine Eingaben),
 - **eine** oder
 - **mehrere Parameterdeklarationen**

besitzen; die **Form** einer **Parameterdeklaration** ist

```
<Typ> <Parameter-Name>
```

Mehrere Deklarationen werden jeweils einzeln definiert und durch **Komma** getrennt

Bsp.: 2 Parameter vom Typ **double** und 1 Parameter von Typ **int**

```
... (double x, double y, int a) { ...
```

Hinweis: Zusammenfassungen sind nicht erlaubt!

```
... (double x, y, int a) { ...
```

▪ Der **Rumpf einer Methode** besteht aus

- Deklarationen **lokaler Variablen** und **Konstanten** (**final**) – diese müssen vor der ersten Verwendung initialisiert werden

Hinweis: Die **(formalen) Parameter** (sofern die Parameterliste nicht leer ist) werden wie lokale Variable verwendet; die Initialisierung dieser Größen mit Werten erfolgt durch die Übergabe der aktuellen Parameter

- Die **Anweisungen** legen die Funktionalität der Methode fest (das ist das Innere der *Black Box*); die Anweisungen können selbst aus
 - Einzelanweisungen,
 - zusammengesetzten Anweisungen,
 - Wiederholungsanweisungen,
 - Programmverzweigungen und
 - Aufrufen von Methoden

bestehen (gleicher Aufbau wie die Methode `main`)

Beispiele für Deklarationen

- **playGame**: Methoden ohne Parameter und ohne Rückgabewert, <Modifizierer> sind **public** und **static**

```
public static void playGame() {  
    ...  
}
```

- **getNextNumber**: Methode mit einem Parameter `number` vom Typ **int** und Rückgabewert vom Typ **int**, keine <Modifizierer>

```
int getNextNumber(int number) {  
    ...  
}
```

- **lessThan**: Methode mit zwei Parametern `x` und `y` jeweils vom Typ **double** und Rückgabewert vom Typ **boolean**, <Modifizierer> ist **static**

```
static boolean lessThan(double x, double y) {  
    ...  
}
```

Zur Rolle der Modifizierer

- Methoden ermöglichen es, den Zustand von Objekten zu verändern oder anzuzeigen; die <Modifizierer> legen Eigenschaften der Methoden fest, z.B.
 - von wo sie **jeweils aufgerufen** werden können (nur innerhalb der Klasse, in der sie deklariert wurden oder auch von Methoden in anderen Klassen)
 - ob sie mit der Instanz einer Klasse (als Objekt) oder einzig durch die Klassendefinition **existieren** (**static**; dieser Punkt ist insbesondere Detail der objekt-orientierten Programmierung, **Teil VII**)
- Beispiele für <Modifizierer>: **public** und **private** spezifizieren den Zugriff (*access specifier*) für die Methode
 - **public**: die Methode kann von überall aus aufgerufen werden, auch außerhalb der Klasse (**class**) in der sie definiert wurde
 - **private**: die Methode kann nur innerhalb derselben Klasse aufgerufen werden, in der sie definiert wurde

Hinweis: Wird kein Zugriff spezifiziert, dann ist die *default*-Regel, dass die Methode überall in dem Paket (package) aufgerufen werden kann, das die Klasse enthält, in der die Methode spezifiziert wurde

Aufruf von Methoden

Allgemeines Format für eine statische Methode

- Aufruf aus **derselben** Klasse

```
<Methoden-Name> (<Parameter>)
```

- Aufruf von **außerhalb** der Klasse (in der die Methode definiert wurde)

```
<Klassen-Name>.<Methoden-Name> (<Parameter>)
```

Hinweis: Die Liste der <Parameter> kann leer sein, z.B. `playGame()`; die **Klammern ()** müssen sowohl bei der **Deklaration** als auch beim **Aufruf immer angegeben** werden!

Beispiele für Aufrufe von Methoden

Einordnung: In den **vorherigen Abschnitten** hatten wir **Ein-/Ausgabebefehle** verwendet sowie Konzepte zum **strukturierten Entwurf** von Programmen kennen gelernt – hierbei wurden bereits Methoden (Unterprogramme) aufgerufen

- Ausgabe von Text: `System.out.println("Hello world ...");`

Klasse

Methoden-Name

Parameter vom Typ
String

- Einlesen einer Zahl: `TextIO.getlnDouble();`

Methoden-Name

Strukturierung durch Methoden – Beispiele

Ratespiel

- Entwurf des Programms durch schrittweise Verfeinerung
 - Das (Haupt-) Programm (`main` Methode) ruft eine Methode (Funktion) auf, die eine Zahl zw. 1 und 100 zufällig wählt; der Benutzer muss diese Zahl raten
 - Die Methode wird so häufig aufgerufen, wie es der Benutzer wünscht

- Grobstruktur:

```
Pick a random number;  
while the game is not over:  
    Get the user's guess;  
    Tell the user whether the guess is  
        high, low, or correct;  
endwhile
```

- Das **Spiel endet** (Termination), wenn der Benutzer richtig geraten hat oder 6-mal hintereinander falsch geraten hat

Pseudocode:

```
Let computerNumber be a random number between 1 and 100;
Let guessCount := 0;
while (true):
    Get the user's guess;
    Count the guess by adding 1 to guessCount;
    if user's guess equals computerNumber:
        Tell the user he has won;
        Break out of the loop;
    endif
    if number of guesses is 6:
        Tell the user he has lost;
        Break out of the loop;
    endif
    if user's guess is less than computersNumber:
        Tell the user the guess was low;
    else if user's guess is higher than computersNumber:
        Tell the user the guess was high;
    endif
endwhile
```

■ Methode in Java

Zufallszahl (Typ Integer) zwischen 1 und 100 mit `Math.random()` liefert zunächst Wert `[0, 1)`

```
static void playGame() {
    int computersNumber, // zufaellige Zahl durch den Computer
        usersGuess,      // Zahl, die der Benutzer eingibt
        guessCount;      // Anzahl der Versuche zaehlen

    computersNumber = (int)(100 * Math.random()) + 1;
    guessCount = 0;
    TextIO.putln();
    TextIO.put("Geben Sie die erste Zahl ein (1. Ratewert) ");
    while (true) {
        usersGuess = TextIO.getInt(); // Versuch des Benutzers
        guessCount++;
        if (usersGuess == computersNumber) {
            TextIO.putln("Sie haben's in " + guessCount +
                " Versuchen geschafft! Meine Zahl war " + computersNumber);
            break; // Spiel ist aus - Benutzer gewinnt
        }
        if (guessCount == 6) {
            TextIO.putln("Verloren! Sie haben's in 6 Versuchen nicht geschafft ... " +
                "Meine Zahl war " + computersNumber);
            break; // Spiel ist aus - Computer gewinnt
        }
        if (usersGuess < computersNumber) // ab hier wird das Spiel fortgesetzt ...
            TextIO.putln("Ratewert zu klein ... noch einmal ... ");
        else if (usersGuess > computersNumber)
            TextIO.putln("Ratewert zu gross ... noch einmal ... ");
    }
    TextIO.putln();
} // end of playGame
```

■ *Wo wird die Methode `playGame()` definiert?*

- a) Die Methode ist in der **selben Klasse** wie die `main()` Methode definiert (später bei der objektorientierten Programmierung dann auch in anderen Klassen)
- b) Die Methode kann **nicht innerhalb** von `main()` oder einer anderen **Methode** definiert werden

Hinweise:

- Blöcke `{ . . . }` dürfen innerhalb von Methoden oder anderen einfachen Blöcken definiert werden – auch mehrfach verschachtelt
- in anderen Programmiersprachen können lokale Funktionen definiert werden, z.B. in MODULA-2

- Die **Reihenfolge** der Definition von Methoden ist beliebig, z.B. `playGame()` kann **vor** oder **nach** der `main()` Methode deklariert werden

■ Vollständiges Java-Programm (Demo: [GuessingGame.java](#))

```

1 public class GuessingGame {
2     /**
3      * Das Programm realisiert ein einfaches Ratespiel in dem der Computer eine
4      * Zahl zwischen 1 und 100 zufaellig zieht. Der Benutzer muss diese Zahl durch
5      * eigene Eingaben raten. Das Programm gibt im Fall einer fehlerhaften Eingabe
6      * einen Hinweis, ob der geratene Wer zu gro\3 oder zu klein war. Fuer den
7      * Ratezyklus hat der Benutzer eine max. Anzahl von Versuchen zur Verfuegung.
8      * Es werden vorgestellt:
9      * - Unterprogramme und Funktionen: Definition, Parameter und Rueckgabewerte
10     * - Verwendung von Wiederholungen (do .. while) und Auswahl (if)
11     * Autor: David J. Eck (mit Ergänzungen hn, Nov.2010)
12     */
13     public static void main(String[] args) {
14         boolean playAgain;

15         TextIO.putln("Wir spielen ein Spiel. Computer waehlt eine Zahl [1, 100)");
16         TextIO.putln("Benutzer versucht, diese zu erraten.");

17         do {
18             playGame(); // Aufruf des Unterprogramms, um ein Spiel zu spielen
19             TextIO.put("Moechten Sie noch ein Spiel spielen? ");
20             playAgain = TextIO.getlnBoolean();
21         } while (playAgain);

22         TextIO.putln("Vielen Dank fuer das Spiel. Tschuess ...");
23     } // end main

24     /**
25     * playGame() fuehrt ein Spiel mit der Rateaufgabe durch
26     * . Eingabe: keine (void)
27     * . Ausgabe: keine (void)
28     */
29     static void playGame() {
30         int computersNumber, // zufaellige Zahl durch den Computer

```

Beispiel für die sinnvolle Verwendung einer **do..while** Schleife

Fortsetzung ...

Vollständiges Java-Programm (Fortsetzung)

```

32 static void playGame() {
33     int computersNumber, // zufaellige Zahl durch den Computer
34     usersGuess,         // Zahl, die der Benutzer eingibt
35     guessCount;         // Anzahl der Versuche zaehlen
36     final int MAX_GUESSES = 6; // maximal 6x raten ...
37
38     computersNumber = (int)(100 * Math.random()) + 1;
39     guessCount = 0;
40     TextIO.putln();
41     TextIO.put("Geben Sie die erste Zahl ein (1. Ratewert) ");
42     while (true) {
43         usersGuess = TextIO.getInt(); // Versuch des Benutzers
44         guessCount++;
45         if (usersGuess == computersNumber) {
46             TextIO.putln("Sie haben's in " + guessCount +
47                 " Versuchen geschafft! Meine Zahl war " + computersNumber);
48             break; // Spiel ist aus - Benutzer gewinnt
49         }
50         if (guessCount == MAX_GUESSES) {
51             TextIO.putln("Verloren! Sie haben's in " + MAX_GUESSES +
52                 " Versuchen nicht geschafft ... " +
53                 "Meine Zahl war " + computersNumber);
54             break; // Spiel ist aus - Computer gewinnt
55         }
56         if (usersGuess < computersNumber) // ab hier wird das Spiel fortgesetzt ...
57             TextIO.putln("Ratewert zu klein ... noch einmal ... ");
58         else if (usersGuess > computersNumber)
59             TextIO.putln("Ratewert zu gross ... noch einmal ... ");
60     }
61     TextIO.putln();
62 } // end of playGame
63 } // end class GuessingGame

```

- Zum allgemeinen Aufbau von Java-Programmen (als Klasse)

```
import ...; // Methoden und Variablen externer Klassen
           // die hier verwendet werden sollen ...
```

```
public class <Klassen-Name> {
```

Klasse (Dateiname: <Klassen-Name>.java;
enthält Methoden und Variablen)

```
    public static void main(String[] args) {
        <Anweisungen>
    }
```

main() Methode: Enthält das
„Hauptprogramm“ mit Anweisungen

```
    static void <Name-1> (...) {
        <Anweisungen>
    }
```

Methode mit Anweisungen (liefert keinen
Ergebniswert)

```
    static int <Name-2> (...) {
        <Anweisungen>
    }
```

Methode mit Anweisungen (liefert
Ergebniswert vom Typ **int**)

```
}
```

Hinweis: Auch hier werden **Blöcke** (`{ ... }`) ineinander geschachtelt; für ihre definierten Größen (Variablen, ...) wird jeweils ein **Gültigkeitsbereich** (**Namensraum**) festgelegt

Lokale Variable und Klassen-Variable

Lokale Variable

- **Variablen** können innerhalb einer Methode deklariert werden, sie sind **nur innerhalb des jeweils umschließenden Blocks** gültig und zugreifbar
- **Lokale Variablen** müssen explizit mit einem Wert **initialisiert** werden, z.B. vor der ersten Verwendung in einem Ausdruck, in **for**-Schleifen, etc.

Bsp.: `char code = 'C';`
`int codeOrd;`

`codeOrd = (int)code;`
`...`

```
if (...) {
    int counter = 0;

    while (counter < MAX) {
        ...
        counter++;
    }
}
```

```
for (int i = 0; i < length; i++) {
    ...
}
```

Klassen-Variable

Einordnung

- Java-Programme sind in Form von **Klassen** aufgebaut und strukturiert
- Man kann **Variablen** deklarieren, die **nicht** Teil einer Methode sind, sondern Teil einer Klasse – sog. **Klassen-Variable**

Bsp.: `public class <Klassen-Name> {`

```
    static String      userName;
    public static int   numberOfPlayers;
    private static double velocity,
                        time;
```

Klassen-Variable sind im Block der Klasse definiert und können aus den Methoden heraus zugegriffen werden

```
    public static void main(...) {
        ...
    }
```

```
    static double <Methoden-Name> (...) {
        ...
    }
```

```
}
```


Deklaration, Zugriff und Gültigkeitsbereich

- **Deklarationen** von **Klassen-Variablen** erfolgen wie diejenigen lokaler Variablen
- Wie Methoden können Klassen-Variable **statisch** oder **nicht-statisch** sein – wir betrachten hier **nur statische Variable** (Modifizierer **static**)
- Statische Klassen-Variable gehören zu der **Klasse, in der sie deklariert wurden** und existieren, **solange ein Programm ausgeführt** wird
- Der **Inhalt** (Wert) einer statischen Klassen-Variable kann von jeder Methode der Klasse gelesen und verändert werden

statische Klassen-Variable werden von allen (statischen) Methoden der Klasse **gemeinsam verwendet** (*shared variable*) – sie definieren **globale Variable**

- Festlegung der **Eigenschaften** von Klassen-Variablen durch **Modifizierer**:
 - **static**: die Klassen-Variable existiert so lange, wie die Klasse ausgeführt wird, d.h. solange der Speicher für die Klasse allokiert ist (wichtig für die objektorientierte Programmierung)
 - **public**: der Inhalt der Klassen-Variable ist auch von einer Methode in einer anderen Klasse aus lesbar und änderbar
 - **private**: die Klassen-Variable kann nur aus Methoden innerhalb derselben Klasse zugegriffen werden

- Klassen-Variable werden bei der **Deklaration automatisch** mit einem **default-Wert initialisiert**
 - numerische Variable mit 0 bei **byte**, **short**, **int**, **long** (oder 0.0 bei **float**, **double**)
 - boolesche Variable mit **false**
 - **char**-Variable mit einem nicht-druckbaren Zeichen, **unicode(0)**
 - String-Variable mit einem speziellen Wert **null**
- **Zugriff** auf Klassen-Variable (vgl. Aufruf von Methoden):
 - Zugriff aus einer Methode **innerhalb** der Klasse

```
<Variablen-Name>
```
 - Zugriff aus einer Methode **außerhalb** der Klasse (wenn die Variable nicht **private** deklariert wurde)

```
<Klassen-Name>.<Variablen-Name>
```
- **Variable**, die außerhalb einer Methode deklariert werden, sind aus Sicht einer solchen Methode **global** (da sie nicht lokal in dem Block der Methode deklariert wurden)

Beispiel – Ratespiel zwischen Programm und Benutzer

- Einführung einer Zähl-Variable `gamesWon`, mit der die gewonnenen Spiele protokolliert werden – und die von jeder Methode aus zugreifbar ist
- Revidierte Version von `GuessingGame` (Demo: [GuessingGame2.java](#))

```

1 public class GuessingGame2 {
2     /**
3      * Das Programm realisiert ein einfaches Ratespiel in dem der Computer eine
4      * Zahl zwischen 1 und 100 zufaellig zieht. Der Benutzer muss diese Zahl durch
5      * eigene Eingaben raten. Das Programm gibt im Fall einer fehlerhaften Eingabe
6      * einen Hinweis, ob der geratene Wer zu gro\3 oder zu klein war. Fuer den
7      * Ratezyklus hat der Benutzer eine max. Anzahl von Versuchen zur Verfuegung.
8      * Diese Version ist eine Erweiterung des Programms 'GuessingGame' in dem hier
9      * in einer Klassen-Variable (globalen Variable) die Anzahl der gewonnenen
10     * Spiele protokolliert wird.
11     * Es werden vorgestellt:
12     * - Unterprogramme und Funktionen: Definition, Parameter und Rueckgabewerte
13     * - Verwendung von Wiederholungen (do .. while) und Auswahl (if)
14     * Autor: David J. Eck (mit Ergaenzungen hn, Nov.2010)
15     */
16     static int gamesWon; // Anzahl der vom Benutzer gewonnenen Spiele (Klassen-Variable)
17
18     public static void main(String[] args) {
19         boolean playAgain;
20
21         gamesWon = 0; // redundante Anweisung
22                     // (die Klassen-Variable wird automatisch mit 0 initialisiert)
23         TextIO.putln("Wir spielen ein Spiel. Computer waehlt eine Zahl [1, 100)");
24         TextIO.putln("Benutzer versucht, diese zu erraten.");
25
26         do {
27             playGame(); // Aufruf des Unterprogramms, um ein Spiel zu spielen
28             TextIO.put("Moechten Sie noch ein Spiel spielen? ");
29             playAgain = TextIO.getlnBoolean();
30         } while (playAgain);
31         TextIO.putln("\n Sie haben " + gamesWon + " Spiele gewonnen.");
32         TextIO.putln("Vielen Dank fuer das Spiel. Tschuess ...");
33     } // end main
34 }

```

Fortsetzung von GuessingGame2

```

34
35  /**
36   * playGame() fuehrt ein Spiel mit der Rateaufgabe durch
37   * . Eingabe: keine (void)
38   * . Ausgabe: keine (void)
39   */
40  static void playGame() {
41      int computersNumber, // zufaellige Zahl durch den Computer
42          usersGuess,      // Zahl, die der Benutzer eingibt
43          guessCount;      // Anzahl der Versuche zaehlen
44      final int MAX_GUESSES = 6; // maximal 6x raten ...
45
46      computersNumber = (int)(100 * Math.random()) + 1;
47      guessCount = 0;
48      TextIO.putln();
49      TextIO.put("Geben Sie die erste Zahl ein (1. Ratewert) ");
50      while (true) {
51          usersGuess = TextIO.getInt(); // Versuch des Benutzers
52          guessCount++;
53          if (usersGuess == computersNumber) {
54              TextIO.putln("Sie haben's in " + guessCount +
55                  " Versuchen geschafft! Meine Zahl war " + computersNumber);
56              gamesWon++;
57              break; // Spiel ist aus - Benutzer gewinnt
58          }
59          if (guessCount == MAX_GUESSES) {
60              TextIO.putln("Verloren! Sie haben's in " + MAX_GUESSES +
61                  " Versuchen nicht geschafft ... " +
62                  "Meine Zahl war " + computersNumber);
63              break; // Spiel ist aus - Computer gewinnt
64          }
65          if (usersGuess < computersNumber) // ab hier wird das Spiel fortgesetzt ...
66              TextIO.putln("Ratewert zu klein ... noch einmal ... ");
67          else if (usersGuess > computersNumber)
68              TextIO.putln("Ratewert zu gross ... noch einmal ... ");
69      }
70      TextIO.putln();
71  } // end of playGame
72 } // end class GuessingGame2

```

Statische Methoden und Variable

Methoden in einzelnen Programmen

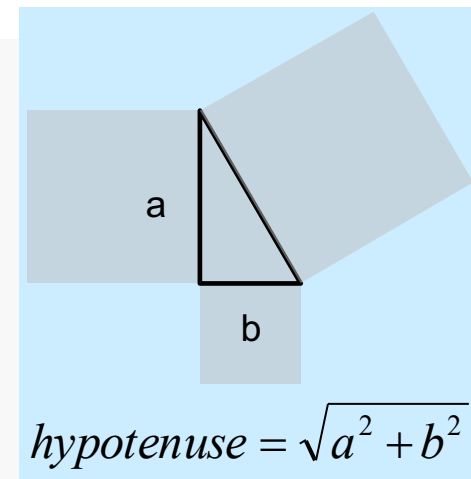
- Die bisher verwendeten Programme (eine Klasse in einer Datei, in der die Methode `main(...)` enthalten ist) verwenden **statische Methoden**

Bsp.:

```
public class Program {
    public static void main(String[] arg) {
        ...
        a = readPosFloat(1);
        ...
        hypotenuse1 = computeRectTriangleHypotenuse(a, b);
    }

    static double readPosFloat(int counter) {
        ...
    }

    static double computeRectTriangleHypotenuse(double length1,
                                                double length2) {
        ...
    }
}
```



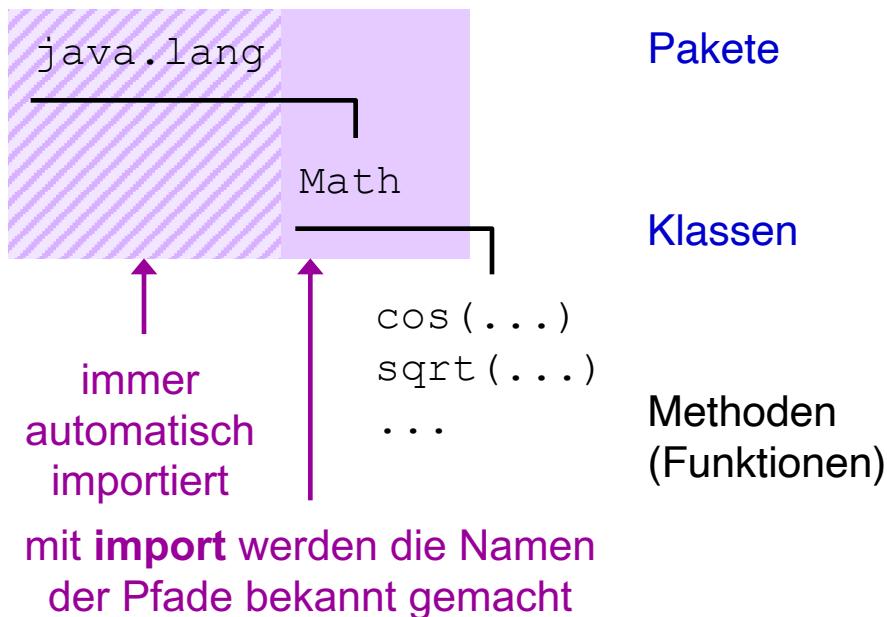
- Die Methoden können **von jeder Stelle** innerhalb von `main(...)` oder den anderen Methoden **aufgerufen** werden

Verwendung statischer Methoden aus anderen Klassen

Hierarchie von Klassen und Paketen – Import von Methoden

- In der **objektorientierten Programmierung** können durch Klassenbildung und Definition von Paketen große Bibliotheken entwickelt werden, in denen verschiedene Methoden und Eigenschaften gruppiert werden
- Auf die Elemente (z.B. statische Methoden, die in einer anderen Klasse definiert wurden) kann über ihre jeweiligen Namen in der Hierarchie zugegriffen werden

Bsp.: Mathematische Methoden/Funktionen (als Teil des Java-Sprachumfangs)



Verwendung:

- Direkt (Auswahl mit **Selektoren**):

```
a = java.lang.Math.cos(...);
```
- Importieren (**import**) von (statischen) Methoden aus Paketen und Klassen:
individuell oder alle:

```
import static java.lang.Math.cos;
a = cos(...);
```



```
import static java.lang.Math.*;
a = cos(...);
```

- Mit der **import**-Anweisung werden die Klassen- oder Methoden-Namen in dem **Namensraum** der aufrufenden Klasse bekannt gemacht; der **Import** dient der Vereinfachung und Strukturierung (zu Namensräumen mehr in **Teil X**)

Bsp.: Folgende Code-Auszüge sind funktionell äquivalent

```
import static java.lang.Math.sqrt;
```

```
public class RectTriangleHypotenuse1 {
    public static void main(String[] args) {
        ...
        hypotenuse2 = sqrt(c*c + c*c);
        ...
    } // end main
} // end class RectTriangleHypotenuse1
```

```
public class RectTriangleHypotenuse1 {
    public static void main(String[] args) {
        ...
        hypotenuse2 = Math.sqrt(c*c + c*c);
        ...
        hypotenuse2 = java.lang.Math.sqrt(c*c + c*c);
    } // end main
} // end class RectTriangleHypotenuse1
```

Variable in Einzelprogrammen

Lokale und globale Variable

In einem Programm werden Variablen deklariert:

1. **Lokale Variablen** werden **innerhalb einer Methode** (bzw. innerhalb von `main(...)`) definiert; die Deklaration innerhalb einzelner Blöcke ist möglich

Lebenszeit: Während der Laufzeit der Methode oder des Blocks, in dem die Variable gültig ist, kann auf die Variable zugegriffen werden

Zugriff: Nur innerhalb der jeweiligen Methode oder in `main(...)`

Bsp.:

```
static double readPosFloat(int counter) {  
    double length;  
    ...  
    length = TextIO.getlnDouble();  
    ...  
}
```

Einordnung: **Lokale Variablen** sind **nicht-statische Variablen**, da sie jeweils beim Aufruf einer Funktion neu instanziiert und nach Beendigung der Ausführung der Methode zerstört werden

2. **Globale Variablen** werden **außerhalb** aller Methoden und außerhalb von `main(...)` definiert; die Deklaration erfolgt innerhalb der Klasse (meist zu Beginn)

Lebenszeit: Die Variable mit einem eindeutigen Namen existiert einmal pro Klasse **während der gesamten Ausführungszeit** des Programms; die Variable wird durch die Deklaration mit dem **Modifizierer `static`** erzeugt (daher wird sie **statische Variable** oder **statisches Attribut** genannt)

Zugriff: Von jeder Methode oder dem Hauptprogramm `main(...)` kann die Variable zugegriffen werden (lesen, schreiben); eine **Änderung des Wertes wirkt sich auf alle Methoden aus**, in denen die Variable verwendet wird

Bsp.:

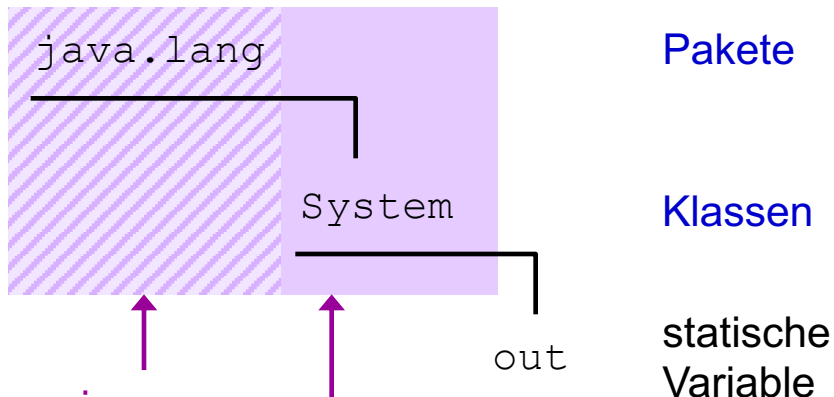
```
public class Program {
    static int counter = 0; // globale Variable

    public static void main(String[] args) {
        int var = 5; // lokale Variable
        ...
        counter = counter + 1;
        ...
    } // end main
} // end class Program
```

Verwendung statischer Variable aus anderen Klassen

- Wie beim Import von Methoden kann auch auf statische Variablen (die in einer anderen Klasse definiert wurden) über ihre jeweiligen Namen in der Hierarchie zugegriffen werden
- Deren Namen werden ebenso mittels der Hierarchie bestehend aus den Paketen und Klassen, in denen sie definiert wurden, bekannt gemacht

Bsp.: Systemausgabe (als Teil des Java-Sprachumfangs)



immer
automatisch
importiert

mit **import** werden die Namen
der Pfade bekannt gemacht

Wie bei den Methoden!

Verwendung:

- Direkt (Auswahl mit **Selektoren**):
`java.lang.System.out.<...>;`
- Importieren (**import**) von (statischen) Variablen aus Paketen und Klassen:
individuell oder alle:


```
import static java.lang.System.*;
out.<...>;
```



```
import static java.lang.*;
System.out.<...>;
```

Gesamteinordnung

- Damit **Methoden** (oder **globale Variablen**) in der vorgestellten Form verwendbar (d.h. zugreifbar, ausführbar) sind, müssen die Deklarationen den Modifizierer **static** enthalten
- Die Möglichkeit der Definition **nicht-statischer** Methoden bzw. Variablen (Attribute) wird in der **objekt-orientierten Programmierung** (**Teil VII**) eingeführt

Hinweis: Der Modifizierer **public** bewirkt, dass eine Methode bzw. globale Variable von Außen, d.h. außerhalb einer Klasse, aufgerufen bzw. verwendet werden kann

```
public class X {
    static public sqrt(double arg) {
        ...
    }
    static public sin(double arg) {
        ...
    }
}
```

```
import static X.sqrt;
import static X.sin;

public class Y {
    public static void main(String[] arg) {
        ...
        val = readInputSin();
        ...
    }

    public static double readInputSin() {
        ...
        return sin(inputVal);
    }
}
```

Lebensdauer von Variablen

- Die Lebensdauer (Existenz) von Variablen ist eine **dynamische** Eigenschaft von Variablen – sie hängt vom Programmablauf ab
- **Programmstart:**
 - (Statische) **Klassen-Variable** werden zu Beginn der Programmausführung angelegt, d.h. es wird Speicherplatz reserviert
 - Sie bekommen einen **default-Wert** zugewiesen
 - ... und bleiben **bis zum Programm-Ende** erhalten (werden erst danach eliminiert)
- Aufruf einer **Methode:**
 - Es werden **formale Parameter** angelegt (Übergabemechanismus: **call-by-value**)
 - Der **Wert** der formalen Parameter ist gleich dem Wert der korrespondierenden **aktuellen Parameter**
 - Bei Ausführung einer Methode werden dessen Anweisungen ausgeführt (inkl. dem Aufruf anderer Methoden oder derselben Methode bei Rekursionen)
 - **lokale Variablen** werden bei Deklaration angelegt, deren Initialwert ist undefiniert oder muss explizit zugewiesen werden
 - die Klassen-Variablen und formalen Parameter sind bereits vorhanden
 - Nach **Beendigung** der Methode (letzte Anweisung oder **return (. . .)**) werden die formalen Parameter und ihre Inhalte **gelöscht und sind nicht mehr zugreifbar**, ebenso die lokalen Variablen und ihre Inhalte

3. Parameterübergabe und Rückgabe von Ergebnissen

- Mechanismen der Parameterübergabe
- Rückgabewerte

Mechanismen der Parameterübergabe

Vorbetrachtungen

Rückblick zur schrittweisen Programmentwicklung

- Wir hatten bei den Methoden zur Abstraktion und Wiederverwendung von Programmteilen bereits Varianten
 - ohne Eingabe und
 - mit Eingaben

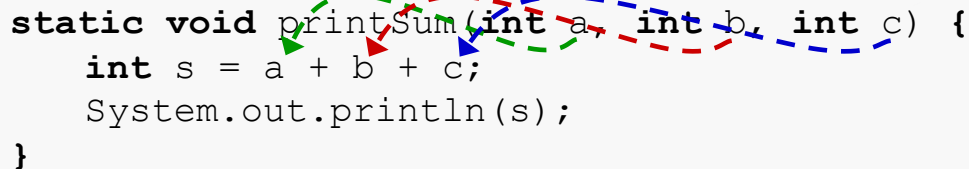
kennen gelernt; die **Eingaben** sind durch die **formalen Parameter** der Methode definiert

- Die Stellen, an denen die **Parameter im Rumpf der Methode verwendet** werden, dienen als Platzhalter für **konkrete Werte (aktuelle Parameter)**
- **Formale Parameter** werden bei einem **Aufruf** zur Verwendung der Methode mit **aktuellen Werten** belegt (Parameter-Substitution); dies erfolgt **wie bei einer Initialisierung lokaler Variablen mit Anfangswerten** (die Werte eines Parameters dürfen gelesen und auch verändert werden)

Zur Parameterübergabe

- Die Parameterübergabe erfolgt **beim Aufruf der Methode**
- Die **Abarbeitung des Methodenaufrufs** geschieht nach folgendem Schema:
 1. Die **aktuellen Parameter** werden **der Reihe nach ausgewertet**; die Parameter können **Variable**, **Konstante** oder Ergebnisse der Auswertung von **Ausdrücken** sein
Hinweis: In Java ist es auch möglich, eine variable Anzahl von Parametern zu übergeben
 2. Die berechneten **Parameterwerte** werden den **formalen Parametern** zugewiesen (Parameterübergabe und Substitution); Bsp.:

Deklaration:



```
static void printSum(int a, int b, int c) {
    int s = a + b + c;
    System.out.println(s);
}
```

Aufruf:

```
...
printSum(10, 13, 20);
...
```

Wichtig: Die Datentypen müssen kompatibel sein

3. Der **Rumpf** der aufgerufenen Methode wird **ausgeführt**, d.h. der Block mit Anweisungen wird gewissermaßen an der Aufrufstelle in den Programmablauf eingeschoben

Beispiel zum **mehrfachen Aufruf** und der Ausführung einer Methode `foo()`:

Programm-Text

```
public class MyClass {
```

```
    static void foo(...) {  
        ...  
    }
```

```
    public static void main(...) {
```

```
        ...
```

```
        foo(...); Aufruf
```

```
        ...
```

```
        foo(...); Aufruf
```

```
        ...
```

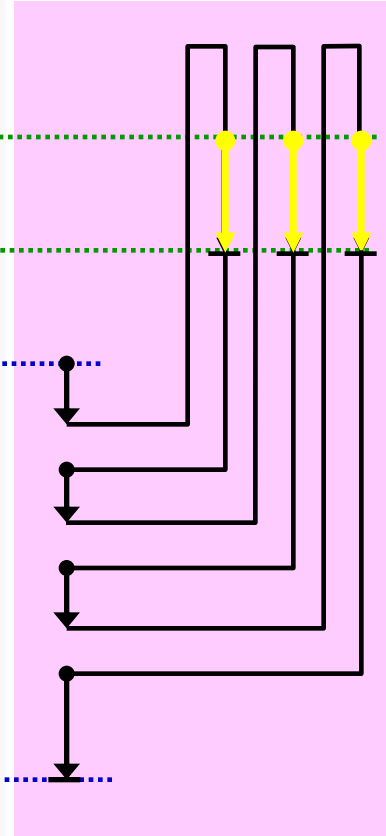
```
        foo(...); Aufruf
```

```
        ...
```

```
    }
```

```
}
```

Programm-Ablauf



Hinweis: Die Methode `foo()` kann auch nach `main()` definiert werden!

4. Die **Ausführung** des Methoden-Rumpfes liefert entweder
- **keinen** Ergebniswert (**void**) oder
 - sie liefert einen Ergebniswert zurück, der dem <Rückgabe-Datentyp> in der Deklaration der Methode entspricht;

die **Rückgabe** wird durch die Steueranweisung **return ...** definiert

Bsp.:

```
static double cube(double argument) {  
    double resCube;  
    resCube = argument * argument * argument;  
    return resCube;  
}
```

5. Der **Ergebniswert** kann **an der Aufrufstelle** in
- einer **Zuweisung** oder
 - einem **Ausdruck**

verwendet werden

Bsp.:

```
val = cube(1.5);  
  
hyp = sin(ang) * sin(ang) + cos(ang) * cos(ang);
```

Substitutionsregeln für Parameter

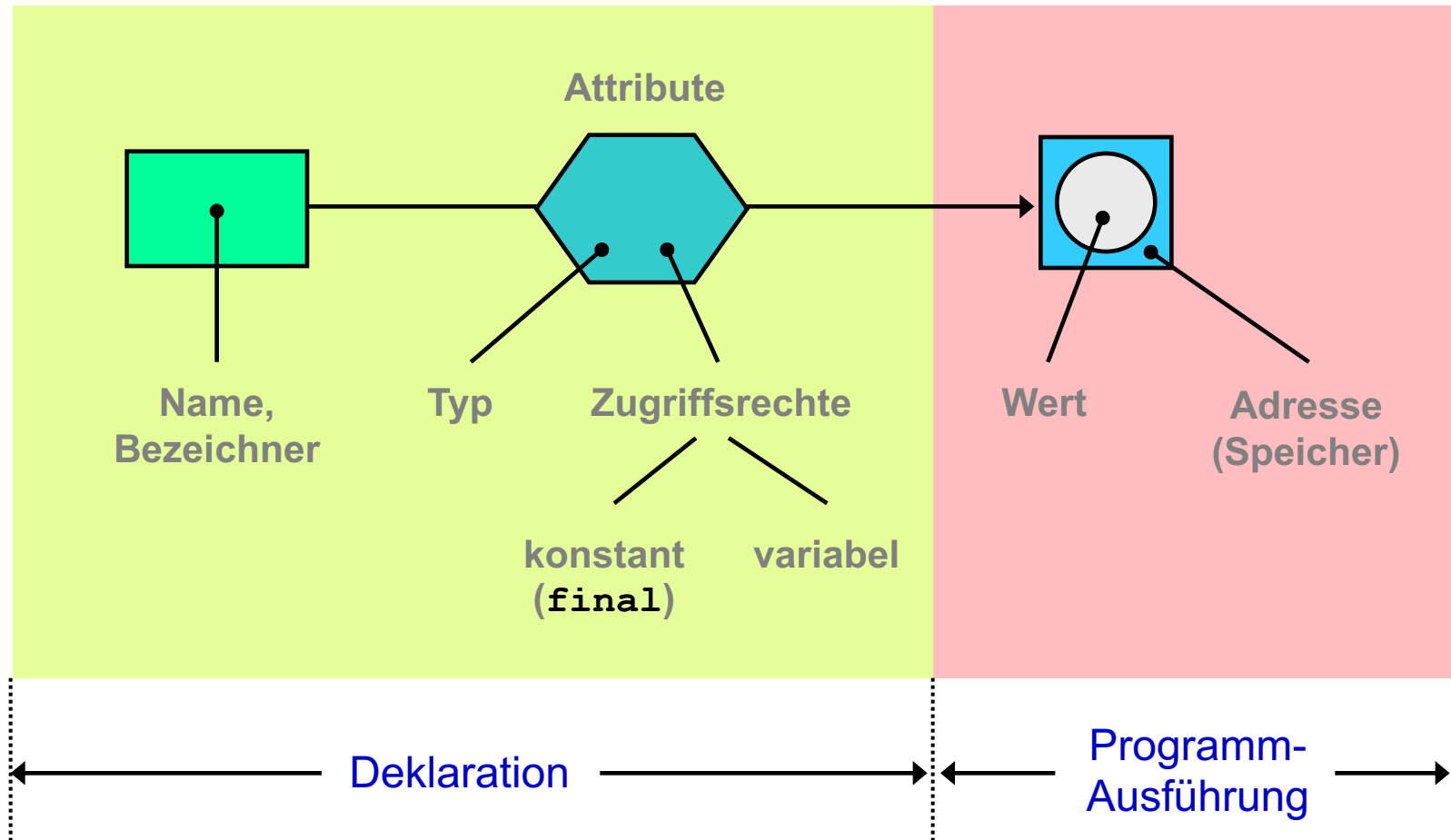
Datenrepräsentation und Substitutionsformen

- Grundsätzliche **Frage**: *Wie gelangen die aktuellen Parameter an der Aufrufstelle im Programm an die in der Methode formal als Platzhalter (formale Parameter) definierten Stellen?*
- Generell gibt es folgende Möglichkeiten der **Parameterübergabe**:
 - **Wertübergabe** (*call-by-value*)
 - **Referenzübergabe** (auch Zeiger- oder Adressübergabe, *call-by-reference*)

Achtung: Die Prinzipien der Parameter-Substitution sind in Programmiersprachen häufig unterschiedlich; wir werden hier

- die Mechanismen der Parameterübergabe bei Java kennen lernen sowie
 - einige ausgewählte Prinzipien bei anderen Programmiersprachen (z.B. MODULA-2, Pascal, C) beschreiben
- In Java werden stets die **Werte** der Parameter, d.h. mittels *call-by-value*, an die aufgerufene Methode übergeben

- Allgemeines Schema der Repräsentation von Datenobjekten



Werte und Adressen (Referenzen, Zeiger)

■ Werte

- Variablen (oder auch Konstante) der einfachen Datentypen assoziieren mittels ihres Namens direkt den Wert



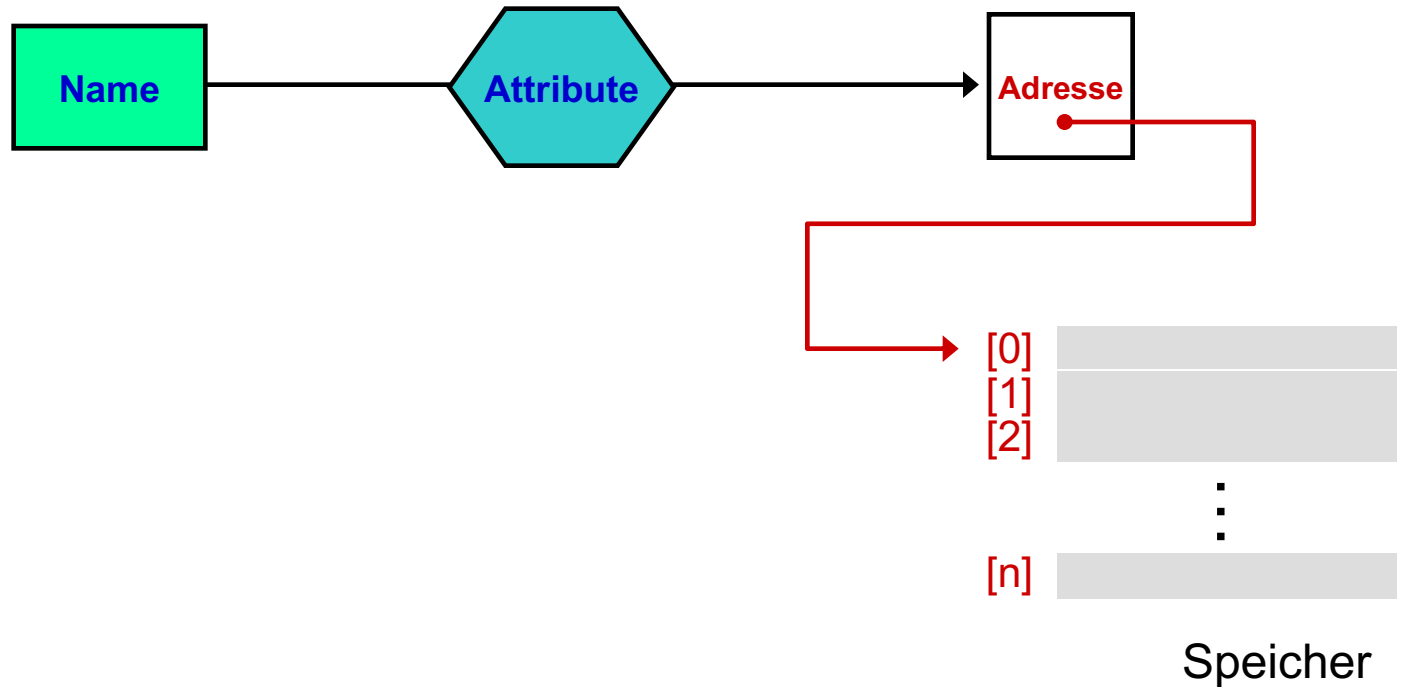
- Der Wert – also das Datum selbst – ist im Speicher (unter einer bestimmten Adresse) abgelegt; diese Adresse der Variablen (oder auch diejenige von Konstanten) ist nicht zugreifbar

■ Adressen (Zeiger) bei Arrays

- Eine Array-Variable ist eine Referenz- oder Zeigervariable und enthält nur die Adresse des Speichersegments, an der das Datenfeld beginnt, nachdem es explizit reserviert wurde (ansonsten ist der Zeiger `null`)



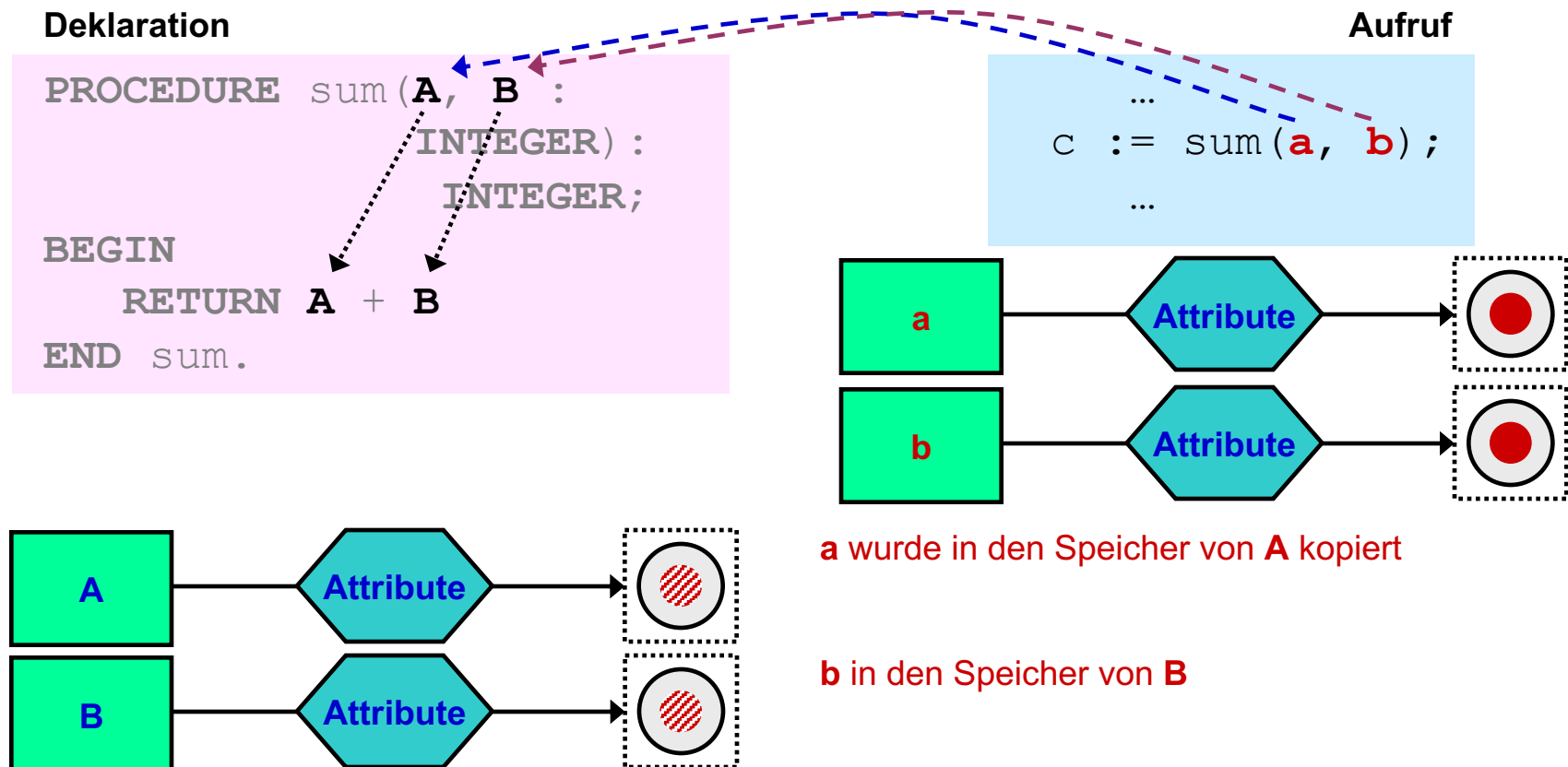
- Der Speicherplatz für eine Referenzvariable wird in Java explizit mit dem Operator **new** angefordert



- Das Aufsuchen eines adressierten Datenobjekts (durch Nachlaufen entlang des Zeigers) nennt man **De-Referenzierung** !

Wertübergabe (*call-by-value*)

- Bei der Wertübergabe werden die **Werte der aktuellen Parameter** in die Platzhalter der formalen Parameter **kopiert** (die wie lokale Variable behandelt werden)
- Beispiel aus **MODULA-2** (Hinweis: andere Syntax als bei Java)



- **Java:** Es wird **ausschließlich** der *call-by-value* Mechanismus verwendet; die **Werte** der aktuellen Parameter werden nach deren Auswertung an die Platzhalter in der aufgerufenen Methode **kopiert**

Deklaration:

```
static int testSum(int s1, int s2) {
    s1 = s1 + 3;
    s2 = s2 * 4;
    return (s1 + s2); // Ergebnis int
}
```

Aufruf:

```
int a = 4,
    b = 5,
    c;

c = testSum(a, b);
System.out.println(a + ", " +
                   b + ", " +
                   c);
```

→ ergibt Ausgabe: **4, 5, 27**

- **Wichtige Eigenschaften**

- Nachdem die Werte der aktuellen Parameter an die formalen Parameter der Methode übergeben wurden, bleiben **alle Änderungen der Variablen innerhalb der Methode** (hier `testSum(...)`) für das aufrufende Programm **verborgen**!
- Nur der **Rückgabewert** (via **return**) wird nach außen geliefert!

Referenz- / Adressübergabe (*call-by-reference*)

- Bei der Referenz-/Adressübergabe werden die **aktuellen Parameter** durch ihre **Adressen** (Speicherorte) identifiziert
- diese Adressen werden an die formalen Parameter gebunden: Im Rumpf der Methode (Unterprogramm) wird eine **Referenz auf den Wert des aktuellen Parameters** benutzt
- Beispiel aus **MODULA-2** (entsprechend bei **PASCAL**)

Deklaration

```
PROCEDURE exch (VAR A, VAR B :  
                INTEGER);
```

```
    buf : INTEGER;
```

```
BEGIN
```

```
    buf := A;
```

```
    A   := B;
```

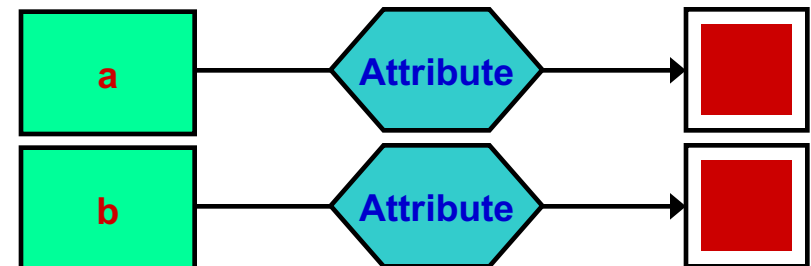
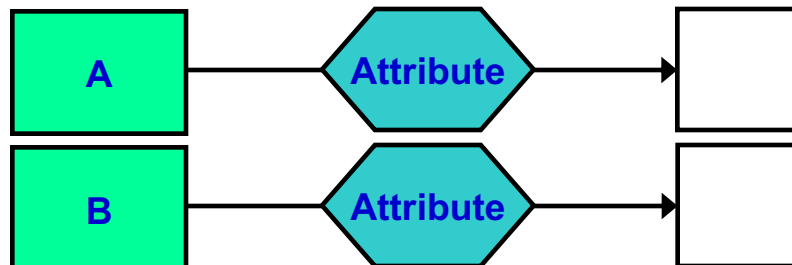
```
    B   := buf;
```

```
END exch.
```

Hinweis: Die Übergabe wird hier durch Angabe des Schlüsselworts **VAR** vor den entsprechenden formalen Parametern gesteuert!

Aufruf

```
...  
exch (a, b);  
...
```



A verweist auf den Speicher von **a**

B verweist auf den Speicher von **b**

■ *Was ist der Unterschied in der Wirkung von Werte- und Referenzübergabe?*

Am Beispiel von **C++** werden die unterschiedlichen Wirkungen noch einmal demonstriert

• *Call-by-value*

Deklaration (C++ Programmfragment):

```
int testSum1(int s1, int s2) {
    s1 = s1 + 3;
    s2 = s2 * 4;
    return (s1 + s2); // Ergebnis int
}
```

Aufruf:

```
int a = 4,
    b = 5,
    c;

c = testSum1(a, b);
std::cout << a << ", " << b <<
", " << c << std::endl;
```

→ ergibt Ausgabe: **4, 5, 27**

• *Call-by-reference*

```
int testSum2(int& s1, int& s2) {
    s1 = s1 + 3;
    s2 = s2 * 4;
    return (s1 + s2); // Ergebnis int
}
```

```
int a = 4,
    b = 5,
    c;

c = testSum2(a, b);
std::cout << a << ", " << b <<
", " << c << std::endl;
```

→ ergibt Ausgabe: **7, 20, 27**

■ *Welche Vorteile bringt der Mechanismus der Referenz-Übergabe?*

Durch die Verwendung einer **Adresse** der zugreifbaren Objekte wird **Aufwand gespart** und somit **Effizienz** gewonnen – gerade **bei großen Datenmengen müssten sonst alle Daten bei der Übergabe zunächst kopiert werden**

Einordnung: Bei Java wird **nur der call-by-value Mechanismus** verwendet; auf den ersten Blick scheint damit der Mechanismus der Adress-Übergabe in Java zu fehlen und damit die Datenübergabe sehr ineffizient zu sein

■ **Genauere Betrachtung:** In Java enthalten die Werte der Variablen ...

- bei **einfachen Datentypen** den **Wert** des Datums selbst,
- bei **dynamischen Datentypen**, mit denen Datenfelder, Zeichenketten, oder Objekte repräsentiert werden, enthält das Datum eine **Referenz** (Zeiger, Adresse) auf das Datum bzw. Objekt

Diese Variablen werden deshalb **Referenzvariable** genannt und es muss Speicherplatz mit dem Operator **new** während der Programmausführung explizit angefordert werden

In anderen Programmiersprachen, in denen der Übergabemechanismus explizit gesteuert wird, sollte die Übergabe jeweils mit Bedacht verwendet werden

Übergabe von Referenzen in Java

- In der Sprache Java wird bei **primitiven Datentypen** automatisch ein **Wertparameter** (*call-by-value*) übergeben
- Bei komplexen Daten (und Objekten), z.B. bei *Arrays*, wird die Referenz als Wertparameter (!) übergeben; die so aufgerufene Methode kann ...
 - den **Wert** der referenzierten Datenmenge bzw. den Zustand des referenzierten Objekts (siehe **Teil VII** – Objektorientierte Programmierung) **verändern**,
 - die **Referenz** selbst **nicht verändern** – letztere zeigt nach der Rückkehr aus dem Unterprogramm (Beendigung der Ausführung nach dem Aufruf) auf dasselbe Datum (bzw. Objekt) wie zuvor
- Beispiel zur **Übergabe einer Referenz auf Arrays** – Programmfragment

```

...
void sortUpArray(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++)
        for (int j = i + 1; j < arr.length; j++)
            if (arr[i] > arr[j]) {
                int buf = arr[i];    ...
                arr[i] = arr[j];    int[] values = new int[] {5, 2, 7, 4};
                arr[j] = buf;        ...
            }
    sortUpArray(values);
} // end sortUpArray                // danach ist das Array, auf das
                                   // 'values' zeigt, sortiert!

```

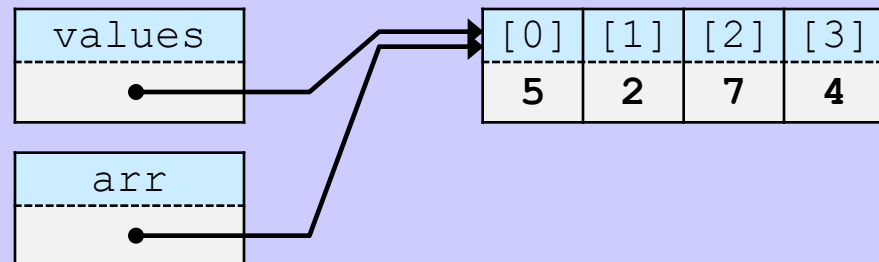
- Schema der **Referenzübergabe** und die Inhaltsänderungen für das Datenfeld (*Array*)

```
void sortUpArray(int[] arr) {
    ...
}

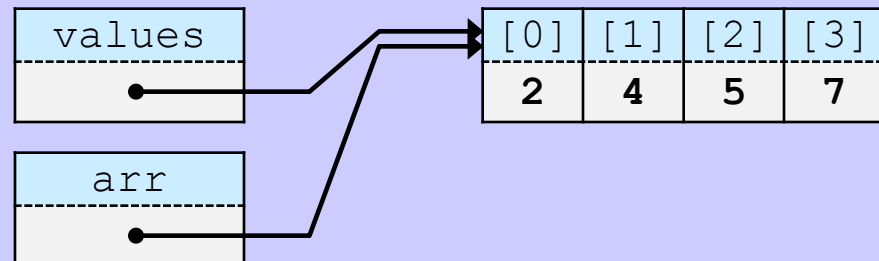
int[] values = new int[] {5, 2, 7, 4};

sortUpArray(values);
```

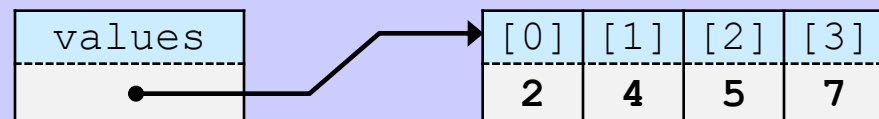
Nach der
Parameterübergabe:



unmittelbar **vor** der Termination
von `sortUpArray`:



unmittelbar **nach** der Termination
von `sortUpArray`:



Der Inhalt der Zeiger-Variable `values` (d.h. der Zeiger auf das *Array*-Objekt) ist nicht verändert, die **Daten des Arrays selbst sind jedoch verändert** (sortiert)

Zusammenfassung: Werte- und Referenz-Parameter

	Wertparameter (<i>call-by-value</i>)	Referenzparameter (<i>call-by-reference</i>)
Formale Parameter	einfache Variablen und strukturierte Variablen (Mechanismus bei <u>Java</u>)	einfache Variablen und strukturierte Variablen
Aktuelle Parameter	beliebige Ausdrücke wie z.B. <code>1.0</code> , <code>2*x</code> , <code>sin(x)</code> , <code>y[i]</code>	nur Variablen, Felder, Feld- elemente, Strukturelemente; <u>keine</u> Konstanten & Ausdrücke
Übergabe	als Kopie (möglicherweise hoher Aufwand, z.B. bei <i>Arrays</i>)	als Adresse übergeben (geringerer Aufwand als bei <i>Arrays</i>)
Zuweisung innerhalb einer Methode	möglich oder verboten (je nach Programmiersprache; in <u>Java</u> ist dies möglich)	möglich
Rückgabe des Wertes bei Methodenende	nein	ja

Unterprogrammaufrufe mit variabler Anzahl von Parametern

Einordnung

- In vielen Programmiersprachen (oder Versionen hiervon) muss die **Anzahl der aktuellen Parameter** mit der **Liste der formalen Parameter übereinstimmen** und in jedem Aufruf angegeben werden (so genannte *fixed arity methods*)
- In Java (seit Version 5.0 oder auch in einigen anderen Sprachen) dürfen in verschiedenen Aufrufen einer Methode eine **unterschiedliche Anzahl von aktuellen Parametern** angegeben werden (*variable arity methods*)

Format und Verwendung

- **Syntax** für variable Parametermengen (Beispiel)

```
public static double average(double... numbers) {
```

Hinweis: “...” (nach Datentyp) bedeutet, dass bei einem Aufruf der Methode eine **beliebige Anzahl von Parametern** des angegebenen Typs übergeben werden können

Bsp.: average(3.14, 2.16, 1.7)

 average(0.375)

 average()

(ebenfalls legaler Aufruf)

 average(1, 2, 3, 4)

(automatische Typkonvertierung)

- Beim **Aufruf** der Methode werden alle **aktuellen Parameter**, die mit dem Datentyp des **variable arity** Parameters korrespondieren, in einem **Array** zusammen gefasst; dieses **Array** wird an die Methode übergeben

Konsequenz: Die **<Dtyp> ...-Parameterliste** wird in der Methode als gewöhnlicher Parameter vom Typ **<Dtyp> []** **behandelt**; die Länge des **Arrays** **<Dtyp> []** gibt an, wie viele aktuelle Parameter übergeben wurden

Bsp.:

```
public static double average(double... numbers) {
    double sum;

    sum = 0.0;
    for (int i = 0; i < numbers.length; i++)
        sum = sum + numbers[i];
    return (sum / numbers.length);
}
```

Hinweise:

- Die **...-Parameterliste** kann **nur als letzter formaler Parameter** in der Definition einer Methode angegeben werden
- Anstelle der Liste individueller Argumente kann auch ein **Array übergeben** werden

Bsp.:

```
double[] salesData;

av = average(salesData); // Mittelwert der Array-Elemente
```

Beispiele

- Aufgrund der automatischen Typ-Konvertierung kann ein *variable arity*-Parameter vom Typ `Object...` einen **aktuellen Parameter** jedes **beliebigen Typs** annehmen
- Formatierte Ausgabe von Werten beliebiger Datentypen

```
public void printf(String format, Object... values) {
```

- Methode zur Konkatination beliebiger Parameter in eine lange Zeichenkette

```
public static String concat(Object... values) {
    String str = ""; // beginne mit leerem String

    for (Object obj : values) { // for each Schleife
        if (obj == null)
            str = str + "null"; // leere Parameterliste
        else
            str = str + obj.toString();
    }
    return str;
} // end concat
```


Rückgabewerte

Methoden mit Rückgabewert (Funktionen)

Schema

- Schema einer **Methode mit Rückgabewert**

```
static <Rueckgabe-Datentyp> <Name> (<Parameter-Liste>) {
    <Anweisungen>
    return <Ausdruck>;
}
```

Ergebnistyp für **return**-Befehl

Rumpf der Methode (mit Ergebnis)

- Eine Methode veranlasst die **Rückgabe eines Ergebniswerts** an die aufrufende Umgebung mittels **return-Anweisung**

```
return <Ausdruck>;
```

- Der Datentyp von **<Ausdruck>** muss zum **<Rueckgabe-Datentyp>** der Methode kompatibel sein; ggf. erfolgt eine implizite Datentypanpassung
- <Datentyp>** können beliebige Java-Typen sein, z.B. **int**, **double**, **String**, ...

Termination von Methoden

- Die Ausführung einer **return**-Anweisung bewirkt die **sofortige Termination** der Methoden-Ausführung und die **unmittelbare Rückkehr zur Aufrufstelle**
- Es können **return**-Anweisungen **an verschiedenen Stellen** im Rumpf einer Methode auftreten

Bsp.:

```
static int match(int[] arr, int elem) {  
    for (int i = 0; i < arr.length; i++) {  
        if (elem == arr[i])  
            return (i); // Element vorhanden (Index)  
    }  
    return (-1); // Element nicht gefunden  
}
```

- Das **Ergebnis** einer Methode (Funktion) kann ...
 - an eine **Variable** zugewiesen werden
 - in einem **Ausdruck** weiter verwendet werden
 - direkt als **Parameter** für eine Methode genutzt werden

Ergebnisse einer Methode ohne Rückgabewert

Methoden ohne Ergebnis

- Bei manchen Methoden ist man nur an der **Ausführung des Rumpfes** interessiert, **nicht** jedoch an **speziellen Ergebniswerten**; in solchen Fällen wird als **<Rueckgabe-Datentyp> void** angegeben („leere“ Rückgabe) und
 - der **<Ausdruck>** in der **return**-Anweisung muss entfallen oder
 - die **return**-Anweisung entfällt insgesamt
- Beispiel einer Methode **ohne Rückgabotyp**

```
void printSize(String label, int value, String unit) {
    // Drucke eine Groesse mit Bezeichnung 'label'
    // (40 Zeichen linksbündig), Wert 'value' (10 Zeichen
    // rechtsbündig) und Einheit 'unit' (linksbündig)

    String empty = "
                ",
           b     = (label + empty + empty).substring(0, 40),
           w     = empty + value;
    w = w.substring(w.length()-10, w.length());
    System.out.println(b + " " + w + " " + unit);
}
```

Seiteneffekte

- Grundsätzlich werden auf **zwei verschiedene Arten** durch die Ausführung von Methoden **Seiteneffekte** erzielt:
 - Wird aus einer Methode auf **Klassen-Variable** (globale Variable) zugegriffen und deren Wert **verändert**, dann bleiben deren Änderungen auch **nach der Termination der Methode** für das Gesamtprogramm weiterhin sichtbar – die Methode hat einen **Seiteneffekt** erzielt
 - Erhält eine Methode eine **Referenz-Variable als Parameter**, so kann die Methode durch **Änderung** der über die Referenz-Variable **adressierten Speicherinhalte** ebenfalls **Seiteneffekte** erzielen (diese werden auch als **kontrollierte Seiteneffekte** bezeichnet)
- Methoden, die Speicherinhalte strukturierter Daten verändern, werden als **void-Methoden** deklariert,
 - da sie keine Werte direkt an die Aufrufstelle liefern,
 - sondern ihre Wirkung durch direkte Manipulation auf den Daten oder Objekten im Speicher erzielen, die nicht lokal mit der Methode assoziiert sind

```

Bsp.: public class TestClass {
    static int number;

    public static void main(String[] args) {
        number = 1;

        System.out.println(number); // Ergebnis: 1
        sideEffect();
        System.out.println(number); // Ergebnis: 5
    } // end main

    static void sideEffect() {
        number = 5;
    } // end sideEffect
} // end class TestClass

```

Bemerkungen:

- Aus Sicht von `main(...)` ist anhand des Aufrufs von `sideEffect()` nicht ersichtlich, dass `number` verändert wird (**unkontrollierter Seiteneffekt**) – **Seiteneffekte** können daher **in komplexen Programmen zu Problemen** führen, da etwaige Fehler hierbei nur schwer zu finden sind
- In der objektorientierten Programmierung (**Teil VII**) werden **Klassen-Variablen über Methoden verändert** – als Prinzip der Informationsverarbeitung (*information hiding*)

Beispiele für Methoden mit Rückgabewert

Berechnung des Bewertungsgrads (A – F) aus einer numerischen Skala

```
static char letterGrade(int numGrade) {
    if (numGrade >= 90)
        return 'A'; // 90 oder darueber ergibt A
    else if (numGrade >= 80)
        return 'B'; // 80 bis 89 ergibt B
    else if (numGrade >= 65)
        return 'C'; // 65 bis 79 ergibt C
    else if (numGrade >= 50)
        return 'D'; // 50 bis 64 ergibt D
    else
        return 'F'; // alles andere ergibt F
} // end letterGrade
```

Texte umdrehen

```
static String reverse(String str) {
    String copy; // Kopie mit umgedrehtem Text

    copy = "";
    for (int i = str.length()-1; i >= 0; i--) { // aktuelle Position in str
        copy = copy + str.charAt(i); // haenge i. Zeichen von str an copy
    }
    return copy;
} // end reverse
```

Berechnung der '3N+1'-Folge

Hier: Berechnung des nächsten Terms der Folge

- Variante mit zwei return-Anweisungen

```
static int nextN(int currentN) {  
    if (currentN % 2 == 1) // ist currentN ungerade?  
        return 3 * currentN + 1;  
    else  
        return currentN / 2;  
} // end nextN
```

- Variante mit einer return-Anweisung

```
static int nextN(int currentN) {  
    int answer; // Wert, der mittels return zurueck gegeben wird  
  
    if (currentN % 2 == 1) // ist currentN ungerade?  
        answer = 3 * currentN + 1;  
    else  
        answer = currentN / 2;  
    return answer;  
} // end nextN
```

- Vollständige Implementierung als Java-Programm (Demo: [ThreeN3.java](#))